

Francisco M. Couto

Data and Text Processing for Health and Life Sciences

OPEN

 Springer

Volume 1137

Advances in Experimental Medicine and Biology

Series Editors

Irun R. Cohen

The Weizmann Institute of Science, Rehovot, Israel

Abel Lajtha

N.S. Kline Institute for Psychiatric Research, Orangeburg, NY, USA

John D. Lambris

University of Pennsylvania, Philadelphia, PA, USA

Rodolfo Paoletti

University of Milan, Milano, Italy

Nima Rezaei

*Tehran University of Medical Sciences, Children's Medical Center Hospital,
Tehran, Iran*

More information about this series at <http://www.springer.com/series/5584>

Francisco M. Couto

Data and Text Processing for Health and Life Sciences

OPEN  Springer

Francisco M. Couto

LASIGE, Department of Informatics, Faculdade de Ciências, Universidade de Lisboa, Lisbon, Portugal

ISSN 0065-2598

e-ISSN 2214-8019

Advances in Experimental Medicine and Biology

ISBN 978-3-030-13844-8

e-ISBN 978-3-030-13845-5

<https://doi.org/10.1007/978-3-030-13845-5>

This book is an open access publication.

© The Editor(s) (if applicable) and The Author(s) 2019



Open Access This book is licensed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this book are included in the book's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the book's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.

The use of general descriptive names, registered names, trademarks, service marks, etc. in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant protective laws and regulations and therefore free for general use.

The publisher, the authors, and the editors are safe to assume that the advice and information in this book are believed to be true and accurate at the date of publication. Neither the publisher nor the authors or the editors give a warranty, express or implied, with respect to the material contained herein or for any errors or omissions that may have been made. The publisher remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

This Springer imprint is published by the registered company Springer Nature Switzerland AG.

The registered company address is: Gewerbestrasse 11, 6330 Cham, Switzerland

*Aos meus pais, Francisco de Oliveira Couto e
Maria Fernanda dos Santos Moreira Couto.*

Preface

During the last decades, I witnessed the growing importance of computer science skills for career advancement in Health and Life Sciences. However, not everyone has the skill, inclination, or time to learn computer programming. The learning process is usually time-consuming and requires constant practice, since software frameworks and programming languages change substantially overtime. This is the main motivation for writing this book about using shell scripting to address common biomedical data and text processing tasks. Shell scripting has the advantages of being: (i) nowadays available in almost all personal computers; (ii) almost immutable for more than four decades; (iii) relatively easy to learn as a sequence of independent commands; (iv) an incremental and direct way to solve many of the data problems that Health and Life professionals face.

During the last decades, I had the pleasure to teach introductory computer science classes to Life and Health and Life Sciences undergraduates. I used programming languages, such as Perl and Python, to address data and text processing tasks, but I always felt to lose a substantial amount of the time teaching the technicalities of these languages, which will probably change over time and are uninteresting for the majority of the students who do not intend to pursue advanced bioinformatics courses. Thus, the purpose of this book is to motivate and help specialists to automate common data and text processing tasks after a short learning period. If they become interested (and I hope some do), the book presents pointers to where they can acquire more advanced computer science skills.

This book does not intend to be a comprehensive compendium of shell scripting commands but instead an introductory guide for Health and Life specialists. This book introduces the commands as they are required to automate data and text processing tasks. The selected tasks have a strong focus on text mining and biomedical ontologies given my research experience and their growing relevance for Health and Life studies. Nevertheless, the same type of solutions presented in the book are also applicable to many other research fields and data sources.

Francisco M. Couto
Lisboa, Portugal
January 2019

Acknowledgments

I am grateful to all the people who helped and encouraged me along this journey, especially to Rita Ferreira for all the insightful discussions about shell scripting.

I am also grateful for all the suggestions and corrections given by my colleague Prof. José Baptista Coelho and by my college students: Alice Veiros, Ana Ferreira, Carlota Silva, Catarina Raimundo, Daniela Matias, Inês Justo, João Andrade, João Leitão, João Pedro Pais, Konil Solanki, Mariana Custódio, Marta Cunha, Manuel Fialho, Miguel Silva, Rafaela Marques, Raquel Chora and Sofia Morais.

This work was supported by FCT through funding of DeST: Deep Semantic Tagger project, ref. PTDC/CCI-BIO/28685/2017 (<http://dest.rd.ciencias.ulisboa.pt/>), and LASIGE Research Unit, ref. UID/CEC/00408/2019.

Acronyms

<i>ChEBI</i>	Chemical Entities of Biological Interest
<i>CSV</i>	Comma-Separated Values
<i>cURL</i>	Client Uniform Resource Locator
<i>DAG</i>	Directed Acyclic Graph
<i>DBMS</i>	Database Management System
<i>DiShIn</i>	Semantic Similarity Measures using Disjunctive Shared Information
<i>DO</i>	Disease Ontology
<i>EBI</i>	European Bioinformatics Institute
<i>GO</i>	Gene Ontology
<i>HTTP</i>	Hypertext Transfer Protocol
<i>HTTPS</i>	HTTP Secure
<i>ICD</i>	International Classification of Diseases
<i>MER</i>	Minimal Named-Entity Recognizer
<i>MeSH</i>	Medical Subject Headings
<i>NCBI</i>	National Center for Biotechnology Information
<i>NER</i>	Named-Entity Recognition
<i>OBO</i>	Open Biological and Biomedical Ontology
<i>OWL</i>	Web Ontology Language
<i>PMC</i>	PubMed Central
<i>RDFS</i>	RDF Schema
<i>SNOMED CT</i>	Systematized Nomenclature of Medicine – Clinical Terms
<i>SQL</i>	Structured Query Language
<i>TSV</i>	Tab-Separated Values
<i>UMLS</i>	Unified Medical Language System
<i>UniProt</i>	Universal Protein Resource
<i>URI</i>	Uniform Resource Identifier
<i>URL</i>	Uniform Resource Locator
<i>XLS</i>	Microsoft Excel file format

XML Extensible Markup Language

Contents

1 Introduction

Biomedical Data Repositories

Scientific Text

Amount of Text

Ambiguity and Contextualization

Biomedical Ontologies

Programming Skills

Why This Book?

Third-Party Solutions

Simple Pipelines

How This Book Helps Health and Life Specialists?

Shell Scripting

Text Files

Relational Databases

What Is in the Book?

Command Line Tools

Pipelines

Regular Expressions

Semantics

2 Resources

Biomedical Text

What?

Where?

How?

Semantics

What?

Where?

How?

Further Reading

3 Data Retrieval

Caffeine Example

Unix Shell

Current Directory

Windows Directories

Change Directory

Useful Key Combinations

Shell Version

Data File

File Contents

Reverse File Contents

My First Script

Line Breaks

Redirection Operator

Installing Tools

Permissions

Debug

Save Output

Web Identifiers

Single and Double Quotes

Comments

Data Retrieval

Standard Error Output

Data Extraction

Single and Multiple Patterns

Data Elements Selection

Task Repetition

Assembly Line

File Header

Variable

XML Processing

Human Proteins

PubMed Identifiers

PubMed Identifiers Extraction

Duplicate Removal

Complex Elements

XPath

Namespace Problems

Only Local Names

Queries

Extracting XPath Results

Text Retrieval

Publication URL

Title and Abstract

Disease Recognition

Further Reading

4 Text Processing

Pattern Matching

Case Insensitive Matching

Number of Matches

Invert Match

File Differences

Evaluation Metrics

Word Matching

Regular Expressions

Extended Syntax

Alternation

Multiple Characters

Quantifiers

Position

Beginning

Ending

Near the End

Word in Between

Full Line

Match Position

Tokenization

Character Delimiters

Wrong Tokens

String Replacement

Multi-character Delimiters

Keep Delimiters

Sentences File

Entity Recognition

Select the Sentence

Pattern File

Relation Extraction

Multiple Filters

Relation Type

Remove Relation Types

Further Reading

5 Semantic Processing

Classes

OWL Files

Class Label

Class Definition

Related Classes

URIs and Labels

URI of a Label

Label of a URI

Synonyms

URI of Synonyms

Parent Classes

Labels of Parents

Related Classes

Labels of Related Classes

Ancestors

Grandparents

Root Class

Recursion

Iteration

My Lexicon

Ancestors Labels

Merging Labels

Ancestors Matched

Generic Lexicon

All Labels

Problematic Entries

Special Characters Frequency

Completeness

Removing Special Characters

Removing Extra Terms

Removing Extra Spaces

Disease Recognition

Performance

Inverted Recognition

Case Insensitive

ASCII Encoding

Correct Matches

Incorrect Matches

Entity Linking

Modified Labels

Ambiguity

Surrounding Entities

Semantic Similarity

Measures

DiShIn Installation

Database File

DiShIn Execution

Large Lexicons

MER Installation

Lexicon Files

MER Execution

Further Reading

Bibliography

Index

1. Introduction

Francisco M. Couto¹

(1) LASIGE, Department of Informatics, Faculdade de Ciências, Universidade de Lisboa, Lisbon, Portugal

Abstract

Health and Life studies are well known for the huge amount of data they produce, such as high-throughput sequencing projects (Stephens et al., PLoS Biol 13(7):e1002195, 2015; Hey et al., The fourth paradigm: data-intensive scientific discovery, vol 1. Microsoft research Redmond, Redmond, 2009). However, the value of the data should not be measured by its amount, but instead by the possibility and ability of researchers to retrieve and process it (Leonelli, Data-centric biology: a philosophical study. University of Chicago Press, Chicago, 2016). Transparency, openness, and reproducibility are key aspects to boost the discovery of novel insights into how living systems work (Nosek et al., Science 348(6242):1422–1425, 2015).

Keywords Bioinformatics – Biomedical data repositories – Text files – EBI: European Bioinformatics Institute – Bibliographic databases – Shell scripting – Command line tools – Spreadsheet applications – CSV: comma-separated values – TSV: tab-separated values

Biomedical Data Repositories

Fortunately, a significant portion of the biomedical data is already being collected, integrated and distributed through Biomedical Data Repositories, such as European Bioinformatics Institute (EBI) and National Center for Biotechnology Information (NCBI) repositories (Cook et al. 2017; Coordinators 2018). Nonetheless, researchers cannot rely on available data as mere facts, they may contain errors, can be outdated, and may require a context (Ferreira et al.

2017). Most facts are only valid in a specific biological setting and should not be directly extrapolated to other cases. In addition, different research communities have different needs and requirements, which change over time (Tomczak et al. 2018).

Scientific Text

Structured data is what most computer applications require as input, but humans tend to prefer the flexibility of text to express their hypothesis, ideas, opinions, conclusions (Barros and Couto 2016). This explains why scientific text is still the preferential means to publish new discoveries and to describe the data that support them (Holzinger et al. 2014; Lu 2011). Another reason is the long-established scientific reward system based on the publication of scientific articles (Rawat and Meena 2014).

Amount of Text

The main problem of analyzing biomedical text is the huge amount of text being published every day (Hersh 2008). For example, 813,598 citations¹ were added in 2017 to MEDLINE, a bibliographic database of Health and Life literature². If we read 10 articles per day, it will take us takes more than 222 years to just read those articles. Figure 1.1 presents the number of citations added to MEDLINE in the past decades, showing the increasing large amount of biomedical text that researchers must deal with.

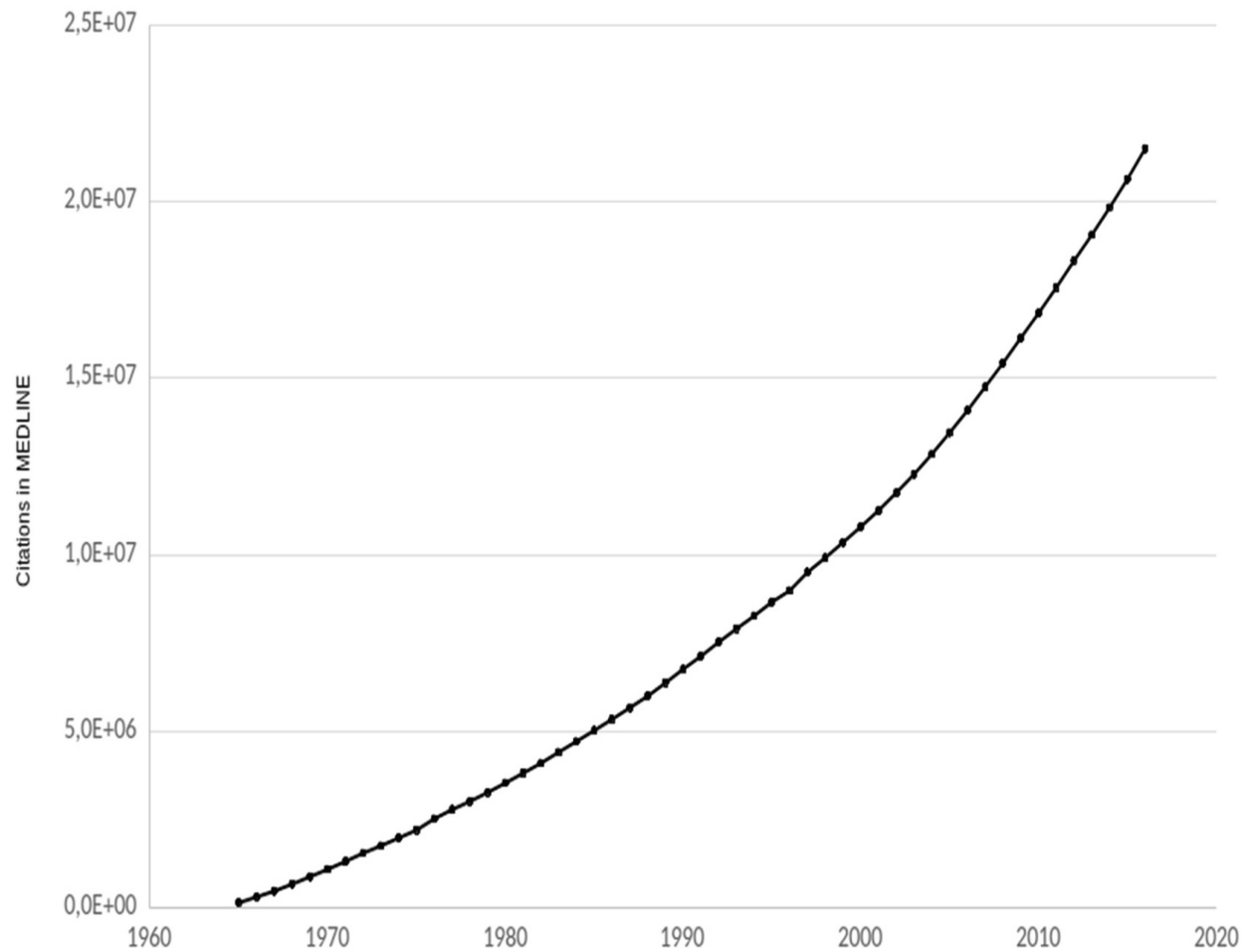


Fig. 1.1 Chronological listing of the total number of citations in MEDLINE (Source: <https://www.nlm.nih.gov/bsd/>)

Moreover, scientific articles are not the only source of biomedical text, for example clinical studies and patents also provide a large amount of text to explore. They are also growing at a fast pace, as Figs. 1.2 and 1.3 clearly show (Aras et al. 2014; Jensen et al. 2012).

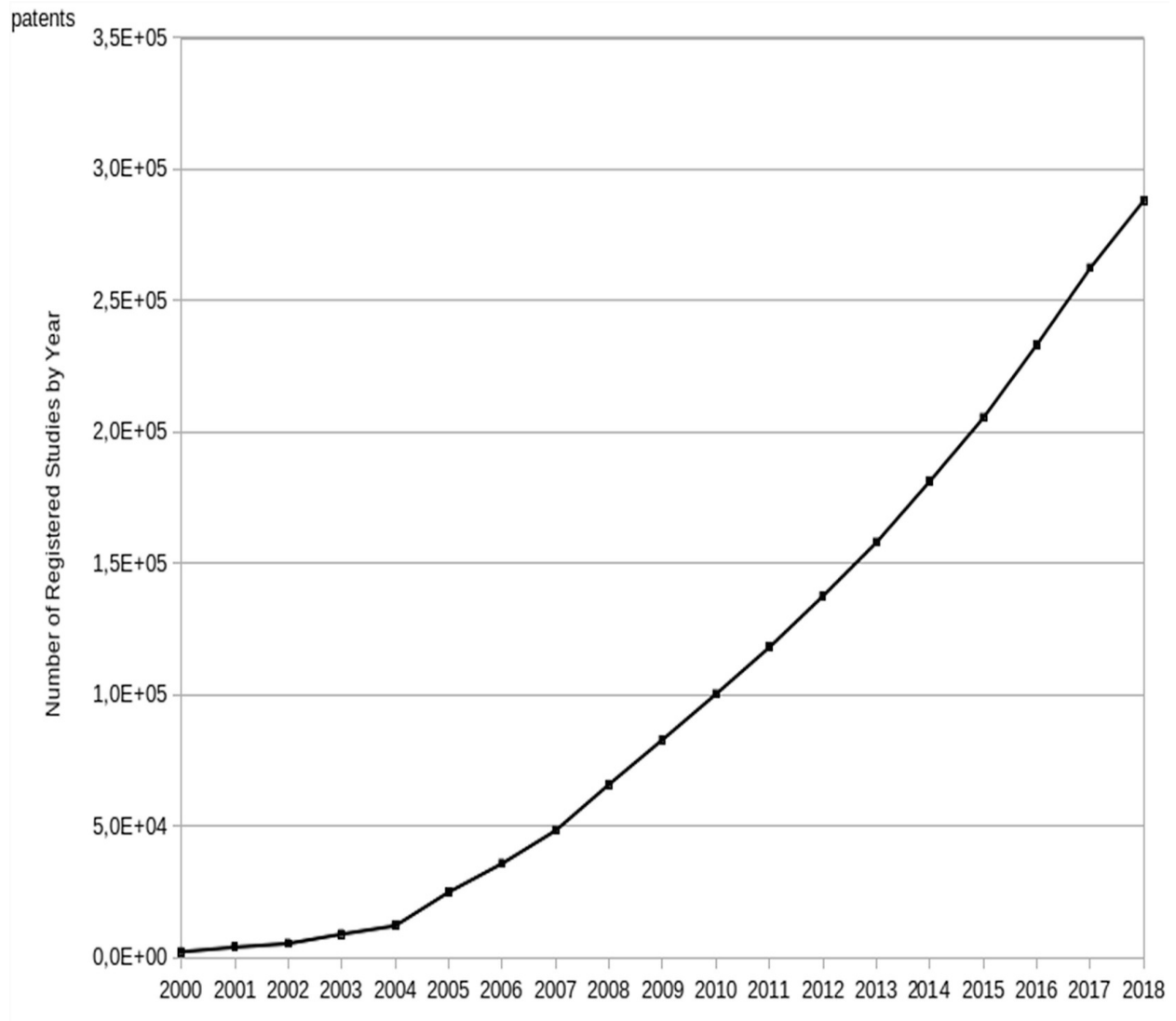


Fig. 1.2 Chronological listing of the total number of registered studies (clinical trials) (Source: <https://clinicaltrials.gov>)

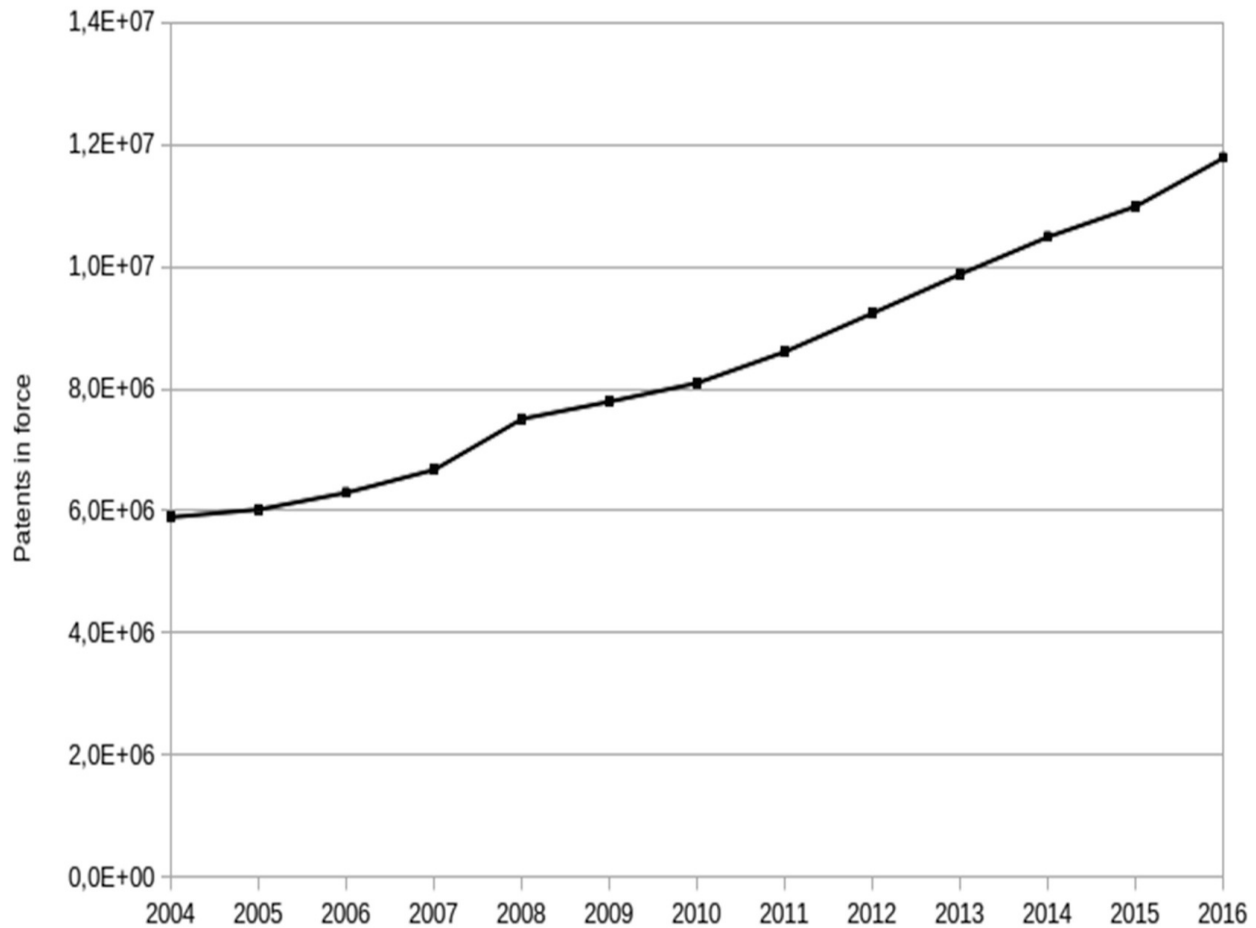


Fig. 1.3 Chronological listing of the total number of patents in force (Source: WIPO statistics database <http://www.wipo.int/ipstats/en/>)

Ambiguity and Contextualization

Given the high flexibility and ambiguity of natural language, processing and extracting information from texts is a painful and hard task, even to humans. The problem is even more complex when dealing with scientific text, that requires specialized expertise to understand it. The major problem with Health and Life Sciences is the inconsistency of the nomenclature used for describing biomedical concepts and entities (Hunter and Cohen 2006; Rebholz-Schuhmann et al. 2005). In biomedical text, we can often find different terms referring to the same biological concept or entity (synonyms), or the same term meaning different biological concepts or entities (homonyms). For example, many times authors improve the readability of their publications by using acronyms to mention entities, that may be clear for experts on the field but ambiguous in another context.

The second problem is the complexity of the message. Almost everyone can read and understand a newspaper story, but just a few can really understand a scientific article. Understanding the underlying message in such articles normally requires years of training to create in our brain a semantic model about the domain and to know how to interpret the highly specialized terminology specific to each domain. Finally, the multilingual aspect of text is also a problem, since most clinical data are produced in the native language (Campos et al. 2017).

Biomedical Ontologies

To address the issue of ambiguity of natural language and contextualization of the message, text processing techniques can explore current biomedical ontologies (Robinson and Bauer 2011). These ontologies can work as vocabularies to guide us in what to look for (Couto et al. 2006). For example, we can select an ontology that models a given domain and find out which official names and synonyms are used to mention concepts in which we have an interest (Spasic et al. 2005). Ontologies may also be explored as semantic models by providing semantic relationships between concepts (Lamurias et al. 2017).

Programming Skills

The success of biomedical studies relies on overcoming data and text processing issues to take the most of all the information available in biomedical data repositories. In most cases, biomedical data analysis is no longer possible using an in-house and limited dataset, we must be able to efficiently process all this data and text. So, a common question that many Health and Life specialists face is:

How can I deal with such huge amount of data and text the necessary expertise, time and disposition to learn computer programming?

This is the goal of this book, to provide a low-cost, long-lasting, feasible and painless answer to this question.

Why This Book?

State-of-the-art data and text processing tools are nowadays based on complex and sophisticated technologies, and to understand them we need to have special

knowledge on programming, linguistics, machine learning or deep learning (Holzinger and Jurisica 2014; Ching et al. 2018; Angermueller et al. 2016). Explaining their technicalities or providing a comprehensive list of them are not the purpose of this book. The tools implementing these technologies tend to be impenetrable to the common Health and Life specialists and usually become outdated or even unavailable some time after their publication or the financial support ends. Instead, this book will equip the reader with a set of skills to process text with minimal dependencies to existing tools and technologies. The idea is not to explain how to build the most advanced tool, but how to create a resilient and versatile solution with acceptable results.

In many cases, advanced tools may not be most efficient approach to tackle a specific problem. It all depends on the complexity of problem, and the results we need to obtain. Like a good physician knows that the most efficient treatment for a specific patient is not always the most advanced one, a good data scientist knows that the most efficient tool to address a specific information need is not always the most advanced one. Even without focusing on the foundational basis of programming, linguistics or artificial intelligence, this book provides the basic knowledge and right references to pursue a more advanced solution if required.

Third-Party Solutions

Many manuscripts already present and discuss the most recent and efficient text mining techniques and the available software solutions based on them that users can use to process data and text (Cock et al. 2009; Gentleman et al. 2004; Stajich et al. 2002). These solutions include stand-alone applications, web applications, frameworks, packages, pipelines, etc. A common problem with these solutions is their resiliency to deal with new user requirements, to changes on how resources are being distributed, and to software and hardware updates. Commercial solutions tend to be more resilient if they have enough customers to support the adaptation process. But of course we need the funding to buy the service. Moreover, we will be still dependent on a third-party availability to address our requirements that are continuously changing, which vary according to the size of the company and our relevance as client.

Using open-source solutions may seem a great alternative since we do not need to allocate funding to use the service and its maintenance is assured by the community. However, many of these solutions derive from academic projects that most of the times are highly active during the funding period and then fade away to minimal updates. The focus of academic research is on creating new and more efficient methods and publish them, the software is normally just a means to demonstrate their breakthroughs. In many cases to execute the legacy software

is already a non-trivial task, and even harder is to implement the required changes. Thus, frequently the most feasible solution is to start from scratch.

Simple Pipelines

If we are interested in learning sophisticated and advanced programming skills, this is not the right book to read. This book aims at helping Health and Life specialists to process data and text by describing a simple pipeline that can be executed with minimal software dependencies. Instead of using a fancy web front-end, we can still manually manipulate our data using the spreadsheet application that we already are comfortable with, and at the same time be able to automatize some of the repetitive tasks.

In summary, this book is directed mainly towards Health and Life specialists and students that need to know how to process biomedical data and text, without being dependent on continuous financial support, third-party applications, or advanced computer skills.

How This Book Helps Health and Life Specialists?

So, if this book does not focus on learning programming skills, and neither on the usage of any special package or software, how it will help specialists processing biomedical text and data?

Shell Scripting

The solution proposed in this book has been available for more than four decades (Ritchie 1971), and it can now be used in almost every personal computer (Haines 2017). The idea is to provide an example driven introduction to shell scripting³ that addresses common challenges in biomedical text processing using a Unix shell⁴. Shells are software programs available in Unix operating systems since 1971⁵, but nowadays are available in most of our personal computers using Linux, macOS or Windows operating systems.

But a shell script is still a computer algorithm, so how is it different from learning another programming language?

It is different in the sense that most solutions are based on the usage of single command line tools, that sometimes are combined as simple pipelines. This book does not intend to create experts in shell scripting, by the contrary, the few scripts introduced are merely direct combinations of simple command line tools individually explained before.

The main idea is to demonstrate the ability of a few command line tools to automate many of the text and data processing tasks. The solutions are presented in a way that comprehending them is like conducting a new laboratory protocol i.e. testing and understanding its multiple procedural steps, variables, and intermediate results.

Text Files

All the data will be stored in text files, which command line tools are able to efficiently process (Baker and Milligan 2014). Text files represent a simple and universal medium of storing our data. They do not require any special encoding and can be opened and interpreted by using any text editor application.

Normally, text files without any kind of formatting are stored using a *txt* extension. However, text files can contain data using a specific format, such as:

CSV : Comma-Separated Values⁶;

TSV
: Tab-Separated Values⁷;

XML
: eXtensible Markup Language⁸.

All the above formats can be open (import), edited and saved (export) by any text editor application. and common spreadsheet applications⁹, such as LibreOffice Calc or Microsoft Excel¹⁰. For example, we can create a new data file using LibreOffice Calc, like the one in Fig. 1.4. Then we select the option to save it as CSV, TSV, XML (Microsoft 2003), and XLS (Microsoft 2003) formats. We can try to open all these files in our favorite text editor.

	A	B
1	A	C
2	G	T
3		

Fig. 1.4 Spreadsheet example

When opening the CSV file, the application will show the following contents:

A, C

G, T

Each line represents a row of the spreadsheet, and column values are separated by commas.

When opening the TSV file, the application will show the following contents:

A C

G T

The only difference is that instead of a comma it is now used a tab character to separate column values.

When opening the XML file, the application will show the following contents:

```

...
<Table ss:StyleID="ta1">
<Column ss:Span="1" ss:Width="
  64.01"/>
<Row ss:Height="12.81"><Cell><
  Data ss:Type="String">A</Data
  ></Cell><Cell><Data ss:Type="
  String">C</Data></Cell></Row>
<Row ss:Height="12.81"><Cell><
  Data ss:Type="String">G</Data
  ></Cell><Cell><Data ss:Type="
  String">T</Data></Cell></Row>
</Table>

```

Now the data is more complex to find and understand, but with a little more effort we can check that we have a table with two rows, each one with two cells.

When opening the XLS file, we will get a lot of strange characters and it is humanly impossible to understand what data it is storing. This happens because XLS is not a text file is a proprietary format¹¹, which organizes data using an exclusive encoding scheme, so its interpretation and manipulation could only be done using a specific software application.

Comma-separated values is a data format so old as shell scripting, in 1972 it was already supported by an IBM product¹². Using CSV or TSV enables us to manually manipulate the data using our favorite spreadsheet application, and at the same time use command line tools to automate some of the tasks.

Relational Databases

If there is a need to use more advanced data storage techniques, such as using a relational database¹³, we may still be able to use shell scripting if we can import and export our data to a text format. For example, we can open a relational database, execute Structured Query Language (SQL) commands¹⁴, and import and export the data to CSV using the command line tool `sqlite3`¹⁵.

Besides CSV and shell scripting being almost the same as they were four decades ago, they are still available everywhere and are able to solve most of our data and text processing daily problems. So, these tools are expected to continue to be used for many more decades to come. As a bonus, we will look like a true professional typing command line instructions in a black background window !



What Is in the Book?

First, the Chap. 2 presents a brief overview of some of the most prominent resources of biomedical data, text, and semantics. The chapter discusses what type of information they distribute, where we can find them, and how we will be able to automatically explore them. Most of the examples in the book use the resources provided by the European Bioinformatics Institute (EBI) and use their services to automatically retrieve the data and text. Nevertheless, after understanding the command line tools, it will not be hard to adapt them to the formats used by other service provider, such as the National Center for Biotechnology Information (NCBI). In terms of semantics, the examples will use two ontologies, one about human diseases and the other about chemical entities of biological interest. Most ontologies share the same structure and syntax, so adapting the solutions to other domains are expected to be painless.

As an example, the Chap. 3 will describe the manual steps that Health and Life specialists may have to perform to find and retrieve biomedical text about *caffeine* using publicly available resources. Afterwards, these manual steps will be automatized by using command line tools, including the automatic download of data. The idea is to go step-by-step and introduce how each command line tool can be used to automate each task.

Command Line Tools

The main command line tools that this book will introduce are the following:

- `curl`: a tool to download data and text from the web;
- `grep`: a tool to search our data and text;
- `gawk`: a tool to manipulate our data and text;
- `sed`: a tool to edit our data and text;
- `xargs`: a tool to repeat the same step for multiple data items;
- `xmllint`: a tool to search in XML data files.

Other command line tools are also presented to perform minor data and text manipulations, such as:

- `cat`: a tool to get the content of file;
- `tr`: a tool to replace one character by another;
- `sort`: a tool to sort multiple lines;

- `head`: a tool to select only the first lines.

Pipelines

A fundamental technique introduced in Chap. 3 is how to redirect the output of a command line tool as input to another tool, or to a file. This enables the construction of pipelines of sequential invocations of command line tools. Using a few commands integrated in a pipeline is really the maximum shell scripting that this book will use. Scripts longer than that would cross the line of not having to learn programming skills.

Chapter 4 is about extracting useful information from the text retrieved previously. The example consists in finding references to *malignant hyperthermia* in these *caffeine* related texts, so we may be able to check any valid relation.

Regular Expressions

A powerful pattern matching technique described in this chapter is the usage of regular expressions¹⁶ in the `grep` command line tool to perform Named-Entity Recognition (NER)¹⁷. Regular expressions originated in 1951 (Kleene 1951), so they are even older than shell scripting, but still popular and available in multiple software applications and programming languages (Forta 2018). A regular expression is a string that include special operators represented by special characters. For example, the regular expression `A | C | G | T` will identify in a given string any of the four nucleobases adenine (A), cytosine (C), guanine (G), or thymine (T).

Another technique introduced is tokenization. It addresses the challenge of identifying the text boundaries, such as splitting a text into sentences. So, we can keep only the sentences that may have something we want. Chapter 4 also describes how can we try to find two entities in the same sentence, providing a simple solution to the relation extraction challenge¹⁸.

Semantics

Instead of trying to recognize a limited list of entities, Chap. 5 explains how can we use ontologies to construct large lexicons that include all the entities of a given domain, e.g. humans diseases. The chapter also explains how the semantics encoded in an ontology can be used to expand a search by adding the ancestors and related classes of a given entity. Finally, a simple solution to the Entity Linking¹⁹ challenge is given, where each entity recognized is mapped to a class in an ontology. A simple technique to solve the ambiguity issue when the

same label can be mapped to more than one class is also briefly presented.



Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.

References

Angermueller C, Pärnamaa T, Parts L, Stegle O (2016) Deep learning for computational biology. *Mol Syst Biol* 12(7):878

[\[Crossref\]](#)

Aras H, Hackl-Sommer R, Schwantner M, Sofean M (2014) Applications and challenges of text mining with patents. In: *IPaMin@ KONVENS*

Baker J, Milligan I (2014) Counting and mining research data with unix. Technical report, The Editorial Board of the *Programming Historian*

Barros M, Couto FM (2016) Knowledge representation and management: a linked data perspective. *Yearb Med Inform* 25(1):178–183

[\[Crossref\]](#)

Campos L, Pedro V, Couto F (2017) Impact of translation on named-entity recognition in radiology texts. *Database* 2017:bax064

Ching T, Himmelstein DS, Beaulieu-Jones BK, Kalinin AA, Do BT, Way GP, Ferrero E, Agapow P-M, Zietz M, Hoffman MM *et al* (2018) Opportunities and obstacles for deep learning in biology and medicine. *J R Soc Interface* 15(141):20170387

[\[Crossref\]](#)

Cock PJ, Antao T, Chang JT, Chapman BA, Cox CJ, Dalke A, Friedberg I, Hamelryck T, Kauff F, Wilczynski B *et al* (2009) Biopython: freely available python tools for computational molecular biology and bioinformatics. *Bioinformatics* 25(11):1422–1423

[\[Crossref\]](#)

Cook CE, Bergman MT, Cochrane G, Apweiler R, Birney E (2017) The european bioinformatics institute in 2017: data coordination and integration. *Nucleic Acids Res* 46(D1):D21–D29

[\[Crossref\]](#)

Coordinators NR (2018) Database resources of the national center for biotechnology information. *Nucleic Acids Res* 46(Database issue):D8

[\[Crossref\]](#)

Couto FM, Silva MJ, Lee V, Dimmer E, Camon E, Apweiler R, Kirsch H, Rebholz-Schuhmann D (2006) GOAnnotator: linking protein go annotations to evidence text. *J Biomed Discov Collab* 1(1):19

[\[Crossref\]](#)

Ferreira JD, Inácio B, Salek RM, Couto FM (2017) Assessing public metabolomics metadata, towards improving quality. *J Integr Bioinform* 14(4):67–72
[\[Crossref\]](#)

Forta B (2018) *Learning regular expressions*. Addison-Wesley Professional, Boston

Gentleman RC, Carey VJ, Bates DM, Bolstad B, Dettling M, Dudoit S, Ellis B, Gautier L, Ge Y, Gentry J *et al* (2004) Bioconductor: open software development for computational biology and bioinformatics. *Genome Biol* 5(10):R80
[\[Crossref\]](#)

Haines N (2017) *Beginning Ubuntu for Windows and Mac users: start your journey into free and open source software*. Apress, Berkeley
[\[Crossref\]](#)

Hersh W (2008) *Information retrieval: a health and biomedical perspective*. Springer Science & Business Media, New York

Holzinger A, Jurisica I (2014) Knowledge discovery and data mining in biomedical informatics: the future is in integrative, interactive machine learning solutions. In: *Interactive knowledge discovery and data mining in biomedical informatics*. Springer, Heidelberg, pp 1–18
[\[Crossref\]](#)

Holzinger A, Schantl J, Schroettner M, Seifert C, Verspoor K (2014) Biomedical text mining: state-of-the-art, open problems and future challenges. In: Holzinger A, Jurisica I (eds) *Interactive knowledge discovery and data mining in biomedical informatics*. Springer, Heidelberg, pp 271–300
[\[Crossref\]](#)

Hunter L, Cohen KB (2006) Biomedical language processing: what's beyond pubmed? *Mol Cell* 21(5):589–594
[\[Crossref\]](#)

Jensen PB, Jensen LJ, Brunak S (2012) Mining electronic health records: towards better research applications and clinical care. *Nat Rev Genet* 13(6):395
[\[Crossref\]](#)

Kleene SC (1951) *Representation of events in nerve nets and finite automata*. Technical report, Rand Project Air Force, Santa Monica

Lamurias A, Ferreira JD, Clarke LA, Couto FM (2017) Generating a tolerogenic cell therapy knowledge graph from literature. *Front Immunol* 8:1656
[\[Crossref\]](#)

Lu Z (2011) PubMed and beyond: a survey of web tools for searching biomedical literature. *Database* 2011:baq036
[\[Crossref\]](#)

Rawat S, Meena S (2014) Publish or perish: where are we heading? *J Res Med Sci* 19(2):87
[\[PubMed\]](#)[\[PubMedCentral\]](#)

Rebholz-Schuhmann D, Kirsch H, Couto F (2005) Facts from text—is text mining ready to deliver? *PLoS Biol* 3(2):e65
[\[Crossref\]](#)

Ritchie DM (1971) Unix programmer's manual. Technical report, Technical report Bell

Robinson PN, Bauer S (2011) Introduction to bio-ontologies. Chapman and Hall/CRC, Boca Raton
[Crossref]

Spasic I, Ananiadou S, McNaught J, Kumar A (2005) Text mining and ontologies in biomedicine: making sense of raw text. Brief Bioinform 6(3):239–251
[Crossref]

Stajich JE, Block D, Boulez K, Brenner SE, Chervitz SA, Dagdigian C, Fuellen G, Gilbert JG, Korf I, Lapp H *et al* (2002) The bioperl toolkit: Perl modules for the life sciences. Genome Res 12(10):1611–1618
[Crossref]

Tomczak A, Mortensen JM, Winnenburg R, Liu C, Alessi DT, Swamy V, Vallania F, Lofgren S, Haynes W, Shah NH *et al* (2018) Interpretation of biological experiments changes with evolution of the gene ontology and its annotations. Sci Rep 8(1):5115
[Crossref]

Footnotes

1 https://www.nlm.nih.gov/bsd/index_stats_comp.html

2 <https://www.nlm.nih.gov/bsd/medline.html>

3 https://en.wikipedia.org/wiki/Shell_script

4 https://en.wikipedia.org/wiki/Unix_shell

5 <https://www.in-ulm.de/~mascheck/bourne/#origins>

6 https://en.wikipedia.org/wiki/Comma-separated_values

7 https://en.wikipedia.org/wiki/Tab-separated_values

8 <https://en.wikipedia.org/wiki/XML>

9 <https://en.wikipedia.org/wiki/Spreadsheet>

10 To save in TSV format using the LibreOffice Calc, we may have to choose CSV format and then select as field delimiter the tab character.

11 https://en.wikipedia.org/wiki/Proprietary_format

12 http://bitsavers.trailingedge.com/pdf/ibm/370/fortran/GC28-6884-0_IBM_FORTRAN_Program_Products_for_OS_and_CMS_General_Information_Jul72.pdf

13 https://en.wikipedia.org/wiki/Relational_database

14 <https://en.wikipedia.org/wiki/SQL>

15 <https://www.sqlite.org/cli.html>

16 https://en.wikipedia.org/wiki/Regular_expression

17 https://en.wikipedia.org/wiki/Named-entity_recognition

18 https://en.wikipedia.org/wiki/Relationship_extraction

19 https://en.wikipedia.org/wiki/Entity_linking

2. Resources

Francisco M. Couto¹

- (1) LASIGE, Department of Informatics, Faculdade de Ciências, Universidade de Lisboa, Lisbon, Portugal

Abstract

The previous chapter presented the importance of text and semantic resources for Health and Life studies. This chapter will describe what kind of text and semantic resources are available, where they can be found, and how they can be accessed and retrieved.

Keywords Biomedical literature – Programmatic access – UniProt citations service – Semantics – Controlled vocabularies – Ontologies – OWL: Web Ontology Language – URI: Uniform Resource Identifier – DAG: Directed Acyclic Graphs – OBO: Open Biomedical Ontologies

Biomedical Text

Text is still the preferential means of publishing novel knowledge in Health and Life Sciences, and where we can expect to find all the information about the supporting data. Text can be found and explored in multiple types of sources, the main being scientific articles and patents (Krallinger et al. 2017). However, less formal texts are also relevant to explore, such as the ones present nowadays in electronic health records (Blumenthal and Tavenner 2010).

What?

In the biomedical domain, we can find text in different forms, such as:

Statement: a short piece of text, normally containing personal remarks or an evidence about a biomedical phenomenon;

Abstract: a short summary of a larger scientific document;

Full-text: the entire text present in a scientific document including scattered text such as figure labels and footnotes.

Statements contain more syntactic and semantic errors than abstracts, since they normally are not peer-reviewed, but they are normally directly linked to data providing useful details about it. The main advantage of using statements or abstracts is the brief and succinct form on which the information is expressed. In the case of abstracts, there was already an intellectual exercise to present only the main facts and ideas. Nevertheless, a brief description may be insufficient to draw a solid conclusion, that may require some important details not possible to summarize in a short piece of text (Schuemie et al. 2004). These details are normally presented in the form of a full-text document, which contains a complete description of the results obtained. For example, important details are sometimes only present in figure labels (Yeh et al. 2003).

One major problem of full-text documents is their availability, since their content may have restricted access. In addition, the structure of the full-text and the format on which is available varies according to the journal in where it was published. Having more information does not mean that all of it is beneficial to find what we need. Some of the information may even induce us in error. For example, the relevance of a fact reported in the Results Section may be different if the fact was reported in the Related Work Section. Thus, the usage of full-text may create several problems regarding the quality of information extracted (Shah et al. 2003).

Where?

Access to biomedical literature is normally done using the internet through PubMed¹, an information retrieval system released in 1996 that allows researchers to search and find biomedical texts of relevance to their studies (Canese 2006). PubMed is developed and maintained by the National Center for Biotechnology Information (NCBI), at the U.S. National Library of Medicine (NLM), located at the National Institutes of Health (NIH). Currently, PubMed provides access to more than 28 million citations from MEDLINE, a bibliographic database with references to a comprehensive list of academic journals in Health and Life Sciences². The references include multiple metadata about the documents, such as: title, abstract, authors, journal, publication date. PubMed does not store the full-text documents, but it provides links where we may find the full-text. More recently, biomedical references are also accessible

using the European Bioinformatics Institute (EBI) services, such as Europe PMC³, the Universal Protein Resource (UniProt) with its UniProt citations service⁴.

Other generic alternative tools have been also gaining popularity for finding scientific texts, such as Google Scholar⁵, Google Patents⁶, ResearchGate⁷ and Mendeley⁸.

More than just text some tools also integrate semantic links. One of the first search engines for biomedical literature to incorporate semantics was GOPubMed⁹, that categorized texts according to Gene Ontology terms found in them (Doms and Schroeder 2005). These semantic resources will be described in a following section. A more recent tool is PubTator¹⁰ that provides the text annotated with biological entities generated by state-of-the-art text-mining approaches (Wei et al. 2013).

There is also a movement in the scientific community to produce Open Access Publications, making full-texts freely available with unrestricted use. One of the main free digital archives of free biomedical full-texts is PubMed Central¹¹ (PMC), currently providing access to more than 5 million documents.

Other relevant source of biomedical texts is the electronic health records stored in health institutions, but the texts they contain are normally directly linked to patients and therefore their access is restricted due to ethical and privacy issues. As example, the THYME corpus¹² includes more than one thousand de-identified clinical notes from the Mayo Clinic, but is only available for text processing research under a data use agreement (DUA) with Mayo Clinic (Styler IV et al. 2014).

From generic texts we can also sometimes find relevant biomedical information. For example, some recent biomedical studies have been processing the texts in social networks to identify new trends and insights about a disease, such as processing tweets to predict flu outbreaks (Aramaki et al. 2011).

How?

To automatically process text, we need programmatic access to it, this means that from the previous biomedical data repositories we can only use the ones that allow this kind of access. These limitations are imposed because many biomedical documents have copyright restrictions hold by their publishers. And some restrictions may define that only manual access is granted, and no programmatic access is allowed. These restrictions are normally detailed in the terms of service of each repository. However, when browsing the repository if we face a CAPTCHA challenge to determine whether we are humans or not,

probably means that some access restrictions are in place.

Fortunately, NCBI¹³ and EBI¹⁴ online services, such as PubMed, Europe PMC, or UniProt Citations, allow programmatic access (Li et al. 2015). Both institutions provide Web APIs¹⁵ that fully document how web services can be programmatically invoked. Some resources can inclusively be accessed using RESTful web services¹⁶ that are characterized by a simple uniform interface that make any Uniform Resource Locator (URL) almost self-explanatory (Richardson and Ruby 2008). The same URL shown by our web browser is the only thing we need to know to retrieve the data using a command line tool.

For example, if we search for *caffeine* using the UniProt Citations service¹⁷, select the first two entries, and click on download, the browser will show information about those two documents using a tabular format.

```
PubMed ID Title Authors/Groups
Abstract/Summary
27702941 Genome-wide association
...
22333316 Modeling caffeine
concentrations ...
```

More important is to check the URL that is now being used:

```
https://www.uniprot.org/
citations/?sort=score&desc=&
compress=no&query=id
:27702941%20OR%20id:22333316&
format=tab&columns=id
```

We can check that the URL has three main components: the scheme (https), the hostname (www.uniprot.org), the service (citations) and the data parameters. The scheme represents the type of web connection to get the data, and usually is one of these protocols: Hypertext Transfer Protocol (HTTP) or HTTP Secure (HTTPS)¹⁸. The hostname represents the physical site where the service is available. The list of parameters depends on the data available from the different services. We can change any value of the parameters (arguments) to get different results. For example, we can replace the two PubMed identifiers by the following one 29029291¹⁹, and our browser will now display the information about this new document:

```
PubMed ID Title Authors/Groups
Abstract/Summary
29029291 Nutrition Influences...
```

The good news is that we can use this link with a command line tool and

automatize the retrieval of the data, including extracting the abstract to process its text.

Semantics

Lack of use of standard nomenclatures across biological text makes text processing a non-trivial task. Often, we can find different labels (synonyms, acronyms) for the same biomedical entities, or, even more problematic, different entities sharing the same label (homonyms) (Rebholz-Schuhmann et al. 2005). Sense disambiguation to select the correct meaning of an expression in a given piece of text is therefore a crucial issue. For example, if we find the disease acronym *ATS* in a text, we may have to figure out if it represents the *Andersen-Tawil syndrome*²⁰ or the *X-linked Alport syndrome*²¹. Further in the book, we will address this issue by using ontologies and semantic similarity between their classes (Couto and Lamurias 2019).

What?

In 1993, Gruber (1993) proposed a short but comprehensive definition of ontology as an:

an explicit specification of a conceptualization

In 1997 and 1998, Borst and Borst (1997) and Studer et al. (1998) refined this definition to:

a formal, explicit specification of a shared conceptualization

A conceptualization is an abstract view of the concepts and the relationships of a given domain. A shared conceptualization means that a group of individuals agree on that view, normally established by a common agreement among the members of a community. The specification is a representation of that conceptualization using a given language. The language needs to be formal and explicit, so computers can deal with it.

Languages

The Web Ontology Language (OWL)²² is nowadays becoming one of the most common languages to specify biomedical ontologies (McGuinness et al. 2004). Another popular alternative is the Open Biomedical Ontology (OBO)²³ format developed by the OBO foundry. OBO established a set of principles to ensure

high quality, formal rigor and interoperability between other OBO ontologies (Smith et al. 2007). One important principle is that OBO ontologies need to be open and available without any constraint other than acknowledging their origin.

Concepts are defined as OWL classes that may include multiple properties. For text processing important properties include the labels that may be used to mention that class. The labels may include the official name, acronyms, exact synonyms, and even related terms. For example, a class defining the disease *malignant hyperthermia* may include as synonym *anesthesia related hyperthermia*. Two distinct classes may share the same label, such as *Andersen-Tawil syndrome* and *X-linked Alport syndrome* that have *ATS* as an exact synonym.

Formality

The representation of classes and the relationships may use different levels of formality, such as controlled vocabularies, taxonomies and thesaurus, that even may include logical axioms.

Controlled vocabularies are list of terms without specifying any relation between them. Taxonomies are controlled vocabularies that include subsumption relations, for example specifying that *malignant hyperthermia* is a *muscle tissue disease*. This *is-a* or subclass relations are normally the backbone of ontologies. We should note that some ontologies may include multiple inheritance, i.e. the same concept may be a specialization of two different concepts. Therefore, many ontologies are organized as a directed acyclic graphs (DAG) and not as hierarchical trees, as the one represented in Fig. 2.1. A thesaurus includes other types of relations besides subsumption, for example specifying that *caffeine* has role *mutagen*.

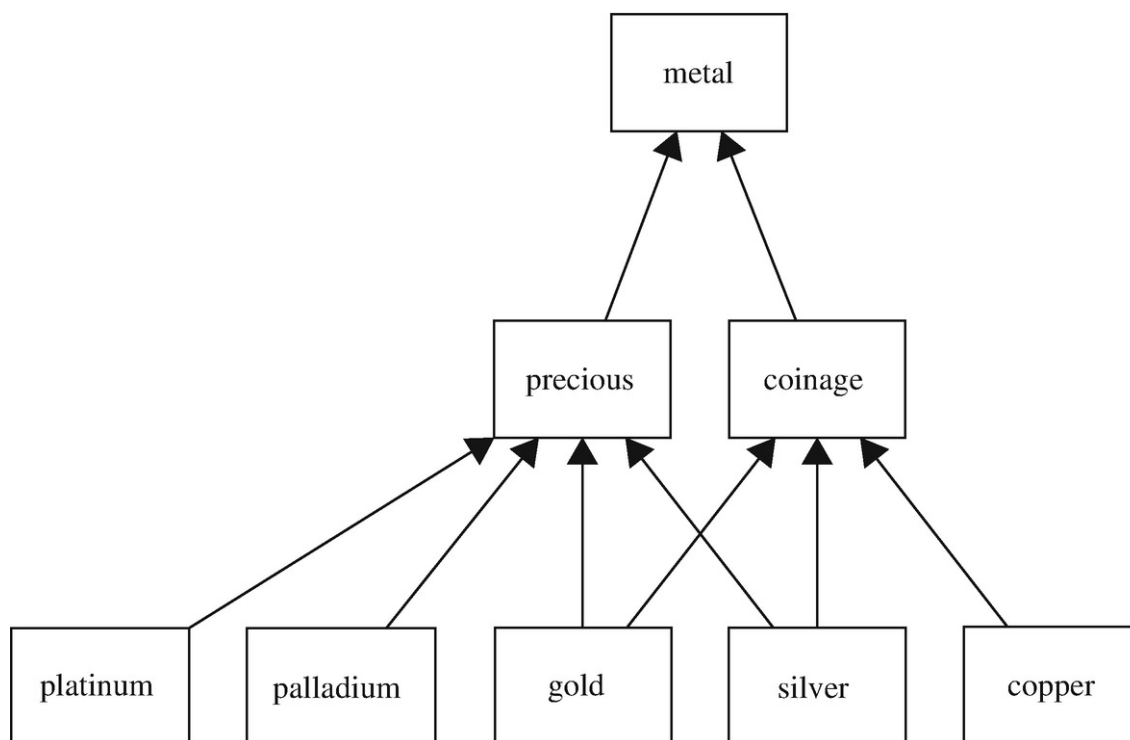


Fig. 2.1 A DAG representing a classification of metals with multiple inheritance, since *gold* and *silver* are considered both precious and coinage metals (All the links represent *is-a* relations)

Gold Related Documents

The importance of these relations can be easily understood by considering the domain modeled by the ontology in Fig. 2.1, and the need to find texts related to *gold*. Assume a corpus with one distinct document mentioning each metal, except for *gold* that no document mentions. So, which documents should we read first?

The document mentioning *silver* is probably the most related since it shares with *gold* two parents, *precious* and *coinage*. However, choosing between the documents mentioning *platinum* or *palladium* or the document mentioning *copper* depends on our information need. This information can be obtained by our previous searches or reads. For example, assuming that our last searches included the word *coinage*, then document mentioning *copper* is probably the second-most related. The importance of these semantic resources is evidenced by the development of the knowledge graph²⁴ by Google to enhance their search engine (Singhal 2012).

Where?

Most of the biomedical ontologies are available through BioPortal²⁵. In December of 2018, BioPortal provided access to more than 750 ontologies

representing more than 9 million classes. BioPortal allows us to search for an ontology or a specific class. For example, if we search for *caffeine*, we will be able to see the large list of ontologies that define it. Each of these classes represent conceptualizations of *caffeine* in different domains and using alternative perspectives. To improve interoperability some ontologies include class properties with a link to similar classes in other ontologies. One of the main goals of the OBO initiative was precisely to tackle this somehow disorderly spread of definitions for the same concepts. Each OBO ontology covers a clearly specified scope that is clearly identified.

OBO Ontologies

A major example of success of OBO ontologies is the Gene Ontology (GO) that has been widely and consistently used to describe the molecular function, biological process and cellular component of gene-products, in a uniform way across different species (Ashburner et al. 2000). Another OBO ontology is the Disease Ontology (DO) that provides human disease terms, phenotype characteristics and related medical vocabulary disease concepts (Schriml et al. 2018). Another OBO ontology is the Chemical Entities of Biological Interest (ChEBI) that provides a classification of molecular entities with biological interest with a focus on small chemical compounds (Degtyarenko et al. 2007).

Popular Controlled Vocabularies

Besides OBO ontologies, other popular controlled vocabularies also exist. One of them is the International Classification of Diseases (ICD)²⁶, maintained by the World Health Organization (WHO). This vocabulary contains a list of generic clinical terms mainly arranged and classified according to anatomy or etiology. Another example is the Systematized Nomenclature of Medicine – Clinical Terms (SNOMED CT)²⁷, currently maintained and distributed by the International Health Terminology Standards Development Organization (IHTSDO). The SNOMED CT is a highly comprehensive and detailed set of clinical terms used in many biomedical systems. The Medical Subject Headings (MeSH)²⁸ is a comprehensive controlled vocabulary maintained by the National Library of Medicine (NLM) for classifying biomedical and health-related information and documents. Both MeSH and SNOMED CT are included in the Metathesaurus of the Unified Medical Language System (UMLS)²⁹, maintained by the U.S National Library of Medicine. This is a large resource that integrates most of the available biomedical vocabularies. The 2015AB release covered more than three million concepts.

Another alternative to BioPortal is Ontobee³⁰, a repository of ontologies used by most OBO ontologies, but it also includes many non-OBO ontologies. In December 2018, Ontobee provided access to 187 ontologies (Ong et al. 2016).

Other alternatives outside the biomedical domain include the list of vocabularies gathered by the W3C SWEO Linking Open Data community project³¹, and by the W3C Library Linked Data Incubator Group³².

How?

After finding the ontologies that cover our domain of interest in the previous catalogs, a good idea is to find their home page and download the files from there. This way, we will be sure that we get the most recent release in the original format and select the subset of the ontology that really matter for our work. For example, ChEBI provides three versions: LITE, CORE and FULL³³. Since we are interested in using the ontology just for text processing, we are probably not interested in chemical data and structures that is available in CORE. Thus, LITE is probably the best solution, and it will be the one we will use in this book. However, we may be missing synonyms that are only included in the FULL version.

OWL

The OWL language is the prevailing language to represent ontologies, and for that reason will be the format we will use in this book. OWL extends RDF Schema (RDFS) with more complex statements using description logic. RDFS is an extension of RDF with additional statements, such as class-subclass or property-subproperty relationships. RDF is a data model that stores information in statements represented as triples of the form subject, predicate and object. Originally, W3C recommended RDF data to be encoded using Extensible Markup Language (XML) syntax, also named RDF/XML. XML is a self-descriptive mark-up language composed of data elements.

For example, the following example represents an XML file specifying that *caffeine* is a drug that may treat the condition of sleepiness, but without being an official treatment:

```
<treatment category="non-  
  official">  
  <drug>caffeine</drug>  
  <condition>sleepiness</  
    condition>  
</treatment>
```

The information is organized in an hierarchical structure of data elements. `treatment` is the parent element of `drug` and `condition`. The character `<` means that a new data element is being specified, and the characters `</` means that a specification of data element will end. The `treatment` element has a property named `category` with the value `non-official`. The `drug` and `condition` elements have as values `caffeine` and `sleepiness`, respectively. This is a very simple XML example, but large XML files are almost unreadable by humans.

To address this issue other encoding languages for RDF are now being used, such as N3³⁴ and Turtle³⁵. Nevertheless, most biomedical ontologies are available in OWL using XML encoding.

URI

The Uniform Resource Identifier (URI) was defined as the standard global identifier of classes in an ontology. For example, the class `caffeine` in ChEBI is identified by the following URI:

```
http://purl.obolibrary.org/obo/
CHEBI_27732
```

If a URI represents a link to a retrievable resource is considered a Uniform Resource Locator, or URL. In other words, a URI is a URL if we open it in a web browser and obtain a resource describing that class.

Sometimes, ontologies are also available as database dumps. These dumps are normally SQL files that need to be fed to a DataBase Management System (DBMS)³⁶. If for any reason we must deal with these files, we can use the simple command line tool named `sqlite3`. The tool has the option to execute the SQL commands to import the data into a database (`.read` command), and to export the data into a CSV file (`.mode` command) (Allen and Owens 2011).

Further Reading

One important read if we need to know more about biomedical resources is the Arthur Lesk's book about bioinformatics (Lesk 2014). The book has entire chapters dedicated to where data and text can be found, providing a comprehensive overview of the type of biomedical information available, nowadays.

A more pragmatic approach is to explore the vast number of manuals, tutorials, seminars and courses provided by the EBI³⁷ and NCBI³⁸.



Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.

References

Allen G, Owens M (2011) The definitive guide to SQLite. Books for professionals by professionals. Apress, Berkeley

Aramaki E, Maskawa S, Morita M (2011) Twitter catches the flu: detecting influenza epidemics using twitter. In: Proceedings of the conference on empirical methods in natural language processing. Association for Computational Linguistics, pp 1568–1576

Ashburner M, Ball CA, Blake JA, Botstein D, Butler H, Cherry JM, Davis AP, Dolinski K, Dwight SS, Eppig JT *et al* (2000) Gene ontology: tool for the unification of biology. *Nat Genet* 25(1):25
[Crossref]

Blumenthal D, Tavenner M (2010) The meaningful use regulation for electronic health records. *N Engl J Med* 363(6):501–504
[Crossref]

Borst W, Borst W (1997) Construction of engineering ontologies for knowledge sharing and reuse. Ph.D. thesis, University of Twente

Canese K (2006) Pubmed celebrates its 10th anniversary. *NLM Tech Bull* 352:e5

Couto F, Lamurias A (2019) Semantic similarity definition. In: Ranganathan S, Nakai K, Schönbach C, Gribskov M (eds) *Encyclopedia of bioinformatics and computational biology*, vol 1. Oxford: Elsevier

Degtyarenko K, De Matos P, Ennis M, Hastings J, Zbinden M, McNaught A, Alcántara R, Darsow M, Guedj M, Ashburner M (2007) ChEBI: a database and ontology for chemical entities of biological interest. *Nucleic Acids Res* 36(suppl_1):D344–D350
[Crossref]

Doms A, Schroeder M (2005) GoPubMed: exploring pubmed with the gene ontology. *Nucleic Acids Res* 33(suppl_2):W783–W786
[Crossref]

Gruber TR (1993) A translation approach to portable ontology specifications. *Knowl Acquis* 5(2):199–220
[Crossref]

Krallinger M, Rabal O, Lourenço A, Oyarzabal J, Valencia A (2017) Information retrieval and text mining technologies for chemistry. *Chem Rev* 117(12):7673–7761
[Crossref]

Lesk A (2014) *Introduction to bioinformatics*. Oxford University Press, Oxford

Li W, Cowley A, Uludag M, Gur T, McWilliam H, Squizzato S, Park YM, Buso N, Lopez R (2015) The embl-ebi bioinformatics web and programmatic tools framework. *Nucleic Acids Res* 43(W1):W580–W584
[Crossref]

McGuinness DL, Van Harmelen F *et al* (2004) OWL web ontology language overview. *W3C Recommendation* 10(10):2004

Ong E, Xiang Z, Zhao B, Liu Y, Lin Y, Zheng J, Mungall C, Courtot M, Ruttenger A, He Y (2016) Ontobee: a linked ontology data server to support ontology term dereferencing, linkage, query and integration. *Nucleic Acids Res* 45(D1):D347–D352
[Crossref]

Rebholz-Schuhmann D, Kirsch H, Couto F (2005) Facts from text—is text mining ready to deliver? *PLoS Biol* 3(2):e65
[Crossref]

Richardson L, Ruby S (2008) RESTful web services. O'Reilly Media, Inc., Sebastopol

Schriml LM, Mittra E, Munro J, Tauber B, Schor M, Nickle L, Felix V, Jeng L, Bearer C, Lichenstein R *et al* (2018) Human disease ontology 2018 update: classification, content and workflow expansion. *Nucleic Acids Res* 47:D955–D962
[Crossref]

Schuemie MJ, Weeber M, Schijvenaars BJ, van Mulligen EM, van der Eijk CC, Jelier R, Mons B, Kors JA (2004) Distribution of information in biomedical abstracts and full-text publications. *Bioinformatics* 20(16):2597–2604
[Crossref]

Shah, P. K., Perez-Iratxeta, C., Bork, P., & Andrade, M. A. (2003). Information extraction from full text scientific articles: where are the keywords? *BMC Bioinformatics* 4(1):20
[Crossref]

Singhal A (2012) Introducing the knowledge graph: things, not strings. Official Google Blog 5. <https://googleblog.blogspot.com/2012/05/introducing-knowledge-graph-things-not.html>

Smith B, Ashburner M, Rosse C, Bard J, Bug W, Ceusters W, Goldberg LJ, Eilbeck K, Ireland A, Mungall CJ *et al* (2007) The obo foundry: coordinated evolution of ontologies to support biomedical data integration. *Nat Biotechnol* 25(11):1251
[Crossref]

Studer R, Benjamins VR, Fensel D *et al* (1998) Knowledge engineering: principles and methods. *Data Knowl Eng* 25(1):161–198
[Crossref]

Styler IV WF, Bethard S, Finan S, Palmer M, Pradhan S, de Groen PC, Erickson B, Miller T, Lin C, Savova G *et al* (2014) Temporal annotation in the clinical domain. *Trans Assoc Comput Ling* 2:143

Wei C-H, Kao H-Y, Lu Z (2013) PubTator: a web-based text mining tool for assisting biocuration. *Nucleic Acids Res* 41(W1):W518–W522
[Crossref]

Yeh A, Hirschman L, Morgan A (2003) Evaluation of text data mining for database curation: lessons learned from the KDD challenge cup. *Bioinformatics* 19(1):i331–i339

Footnotes

1 <https://www.nlm.nih.gov/bsd/pubmed.html>

2 <https://www.nlm.nih.gov/bsd/medline.html>

3 <http://europepmc.org/>

4 <https://www.uniprot.org/citations/>

5 <http://scholar.google.com/>

6 <http://www.google.com/patents>

7 <https://www.researchgate.net/>

8 <https://www.mendeley.com/>

9 <https://pubmed.org/>

10 <http://www.ncbi.nlm.nih.gov/CBBresearch/Lu/Demo/PubTator/>

11 <https://www.ncbi.nlm.nih.gov/pmc/>

12 <http://thyme.healthnlp.org/>

13 <https://www.ncbi.nlm.nih.gov/home/develop/api/>

- 14 <https://www.ebi.ac.uk/seqdb/confluence/display/JDSAT/>
- 15 https://en.wikipedia.org/wiki/Web_API
- 16 <https://www.ebi.ac.uk/seqdb/confluence/pages/viewpage.action?pageId=68165098>
- 17 <https://www.uniprot.org/citations/>
- 18 https://en.wikipedia.org/wiki/Hypertext_Transfer_Protocol
- 19 <https://www.uniprot.org/citations/?sort=score&desc=&compress=no&query=id:29029291&format=tab&columns=id>
- 20 http://purl.obolibrary.org/obo/DOID_0050434
- 21 http://purl.obolibrary.org/obo/DOID_0110034
- 22 https://en.wikipedia.org/wiki/Web_Ontology_Language
- 23 https://en.wikipedia.org/wiki/Open_Biomedical_Ontologies
- 24 https://en.wikipedia.org/wiki/Knowledge_Graph
- 25 <http://bioportal.bioontology.org/>
- 26 <https://www.who.int/classifications/icd/en/>

- 27 <https://digital.nhs.uk/services/terminology-and-classifications/snomed-ct>
- 28 <https://www.nlm.nih.gov/mesh/>
- 29 <https://www.nlm.nih.gov/research/umls/>
- 30 <http://www.ontobee.org/>
- 31 <http://www.w3.org/wiki/TaskForces/CommunityProjects/LinkingOpenData/CommonVocabularies>
- 32 <http://www.w3.org/2005/Incubator/ld/XGR-ld-vocabdataset-20111025>
- 33 <https://www.ebi.ac.uk/chebi/downloadsForward.do>
- 34 <https://en.wikipedia.org/wiki/Notation3>
- 35 [https://en.wikipedia.org/wiki/Turtle_\(syntax\)](https://en.wikipedia.org/wiki/Turtle_(syntax))
- 36 https://en.wikipedia.org/wiki/Database#Database_management_system
- 37 <https://www.ebi.ac.uk/training>
- 38 <https://www.ncbi.nlm.nih.gov/home/learn/>

3. Data Retrieval

Francisco M. Couto¹

- (1) LASIGE, Department of Informatics, Faculdade de Ciências, Universidade de Lisboa, Lisbon, Portugal

Abstract

This chapter starts by introducing an example of how we can retrieve text, where every step is done manually. The chapter will describe step-by-step how we can automatize each step of the example using shell script commands, which will be introduced and explained as long as they are required. The goal is to equip the reader with a basic set of skills to retrieve data from any online database and follow the links to retrieve more information from other sources, such as literature.

Keywords Unix shell – Terminal application – Web retrieval – cURL: Client Uniform Resource Locator – Data extraction – Data selection – Data filtering – Pattern matching – XML: extensible markup language – XPath: XML path language

Caffeine Example

As our main example, let us consider that we need to retrieve more data and literature about *caffeine*. If we really do not know anything about *caffeine*, we may start by opening our favorite internet browser and then searching *caffeine* in Wikipedia¹ to know what it really is (see Fig. 3.1). From all the information that is available we can check in the infobox that there are multiple links to external sources. The infobox is normally a table added to the top right-hand part of a web page with structured data about the entity described on that page.

The image is a screenshot of the Wikipedia page for "Caffeine". The browser's address bar shows the URL "https://en.wikipedia.org/wiki/Caffeine". The page features the Wikipedia logo on the left, a navigation menu with links like "Main page", "Contents", and "Tools", and a search bar at the top right. The article title "Caffeine" is prominently displayed, followed by the text "From Wikipedia, the free encyclopedia". A disclaimer states: "This article is about the stimulant drug. For other uses, see [Caffeine \(disambiguation\)](#)." The main text describes caffeine as a "central nervous system (CNS) stimulant of the methylxanthine class" and provides details about its legal status, mechanisms of action, and chemical classification as a purine and methylxanthine alkaloid. To the right of the text is a chemical structure diagram of caffeine, labeled "Caffeine", which shows the fused ring system with methyl groups and carbonyl groups. Below the diagram is a small 3D ball-and-stick model of the molecule.

Fig. 3.1 Wikipedia page about caffeine

From the list of identifiers (see Fig. 3.2), let us select the link to one resource hosted by the European Bioinformatics Institute (EBI), the link to CHEBI:27732².

	<ul style="list-style-type: none"> 1.2 Enhancing performance 1.3 Specific populations 2 Adverse effects <ul style="list-style-type: none"> 2.1 Physical 2.2 Psychological 2.3 Reinforcement disorders 2.4 Risk of other diseases 3 Overdose 4 Interactions <ul style="list-style-type: none"> 4.1 Alcohol 4.2 Tobacco 4.3 Birth control 4.4 Medications 5 Pharmacology <ul style="list-style-type: none"> 5.1 Pharmacodynamics 5.2 Pharmacokinetics 6 Chemistry <ul style="list-style-type: none"> 6.1 Synthesis 6.2 Decaffeination 6.3 Detection in body fluids 6.4 Analogs 6.5 Precipitation of tannins 7 Natural occurrence 	<table> <tr> <td colspan="2">Excretion</td><td>Urine (100%)</td></tr> <tr> <td colspan="3">Identifiers</td></tr> <tr> <td>IUPAC name</td><td></td><td>[show]</td></tr> <tr> <td>CAS Number</td><td>58-08-2</td><td>✓</td></tr> <tr> <td>PubChem CID</td><td>2519</td><td>✓</td></tr> <tr> <td>IUPHAR/BPS</td><td>407</td><td>✓</td></tr> <tr> <td>DrugBank</td><td>DB00201</td><td>✓</td></tr> <tr> <td>ChemSpider</td><td>2424</td><td>✓</td></tr> <tr> <td>UNII</td><td>3G6A5W338E</td><td>✓</td></tr> <tr> <td>KEGG</td><td>D00528</td><td>✓</td></tr> <tr> <td>ChEBI</td><td>CHEBI:27732</td><td>✓</td></tr> <tr> <td>ChEMBL</td><td>CHEMBL113</td><td>✓</td></tr> <tr> <td>PDB ligand</td><td>CFF (PDB ✓, RCSB PDB ✓)</td><td></td></tr> <tr> <td>ECHA InfoCard</td><td>100.000.329</td><td>✓</td></tr> <tr> <td colspan="3">Chemical and physical data</td></tr> <tr> <td>Formula</td><td colspan="2">C₈H₁₀N₄O₂</td></tr> <tr> <td>Molar mass</td><td colspan="2">194.19 g/mol</td></tr> <tr> <td>3D model (JSmol)</td><td colspan="2">Interactive image</td></tr> <tr> <td>Density</td><td colspan="2">1.23 g/cm³</td></tr> <tr> <td>Melting point</td><td colspan="2">235 to 238 °C (455 to 460 °F) (anhydrous)^{[8][9]}</td></tr> <tr> <td>SMILES</td><td></td><td>[show]</td></tr> <tr> <td>InChI</td><td></td><td>[show]</td></tr> </table>	Excretion		Urine (100%)	Identifiers			IUPAC name		[show]	CAS Number	58-08-2	✓	PubChem CID	2519	✓	IUPHAR/BPS	407	✓	DrugBank	DB00201	✓	ChemSpider	2424	✓	UNII	3G6A5W338E	✓	KEGG	D00528	✓	ChEBI	CHEBI:27732	✓	ChEMBL	CHEMBL113	✓	PDB ligand	CFF (PDB ✓ , RCSB PDB ✓)		ECHA InfoCard	100.000.329	✓	Chemical and physical data			Formula	C ₈ H ₁₀ N ₄ O ₂		Molar mass	194.19 g/mol		3D model (JSmol)	Interactive image		Density	1.23 g/cm ³		Melting point	235 to 238 °C (455 to 460 °F) (anhydrous) ^{[8][9]}		SMILES		[show]	InChI		[show]
Excretion		Urine (100%)																																																																		
Identifiers																																																																				
IUPAC name		[show]																																																																		
CAS Number	58-08-2	✓																																																																		
PubChem CID	2519	✓																																																																		
IUPHAR/BPS	407	✓																																																																		
DrugBank	DB00201	✓																																																																		
ChemSpider	2424	✓																																																																		
UNII	3G6A5W338E	✓																																																																		
KEGG	D00528	✓																																																																		
ChEBI	CHEBI:27732	✓																																																																		
ChEMBL	CHEMBL113	✓																																																																		
PDB ligand	CFF (PDB ✓ , RCSB PDB ✓)																																																																			
ECHA InfoCard	100.000.329	✓																																																																		
Chemical and physical data																																																																				
Formula	C ₈ H ₁₀ N ₄ O ₂																																																																			
Molar mass	194.19 g/mol																																																																			
3D model (JSmol)	Interactive image																																																																			
Density	1.23 g/cm ³																																																																			
Melting point	235 to 238 °C (455 to 460 °F) (anhydrous) ^{[8][9]}																																																																			
SMILES		[show]																																																																		
InChI		[show]																																																																		

Fig. 3.2 Identifiers section of the Wikipedia page about caffeine

CHEBI represents the acronym of the resource Chemical Entities of Biological Interest (ChEBI)³ and 27732 the identifier of the entry in ChEBI describing *caffeine* (see Fig. 3.3). ChEBI is a freely available database of molecular entities with a focus on “small” chemical compounds. More than a simple database, ChEBI also includes an ontology that classifies the entities according to their structural and biological properties.

The screenshot displays the ChEBI entry for caffeine (CHEBI:27732) on the EMBL-EBI website. The page features a search bar at the top with examples like 'iron*', 'InChI=1S/CH4O/c1-2/h2H,1H3', and 'caffeine'. Below the search bar, there are navigation tabs: Home, Advanced Search, Browse, Documentation, Download, Tools, and About ChEBI. The main content area is titled 'ChEBI:27732 - caffeine' and includes a 'Main' tab, 'ChEBI Ontology', 'Automatic Xrefs', 'Reactions', 'Pathways', and 'Models'. The chemical structure of caffeine is shown on the left. To the right, a table provides detailed information:

ChEBI Name	caffeine
ChEBI ID	CHEBI:27732
Definition	A trimethylxanthine in which the three methyl groups are located at positions 1, 3, and 7. A purine alkaloid that occurs naturally in tea and coffee.
Stars	☆☆☆ This entity has been manually annotated by the ChEBI Team.
Secondary ChEBI IDs	CHEBI:3295, CHEBI:41472, CHEBI:22982
Supplier Information	ZINC000000001084 , eMolecules:493944 , eMolecules:27517656
Download	Molfile XML SDF

At the bottom, there is a link: 'Find compounds which contain this structure'.

Fig. 3.3 ChEBI entry describing caffeine

By analyzing the CHEBI:27732 web page we can check that ChEBI provides a comprehensive set of information about this chemical compound. But let us focus on the *Automatic Xrefs* tab⁴. This tab provides a set of external links to other resources describing entities somehow related to *caffeine* (see Fig. 3.4).

caffeine (CHEBI:277) x

European Bioinformatics Institute [GB] | <https://www.ebi.ac.uk/chebi/displayAutoXrefs.do?chebiid=CHEBI:27732>

Main ChEBI Ontology Automatic Xrefs Reactions Pathways Models

+ Expand relevant - Collapse all

Protein Sequences 77

UniProt KB 77
UniProt Knowledge Base of protein sequences.

1. [A2AGL3](#)
Ryanodine receptor 3
2. [A4GE69](#)
7-methylxanthosine synthase 1
3. [A4GE70](#)
3,7-dimethylxanthine N-methyltransferase
4. [A6MFK9](#)
Cysteine-rich venom protein
5. [B0LPN4](#)
Ryanodine receptor 2

[show all >>](#)

Small molecules 21

NMRShiftDB 1
NMRShiftDB is a NMR database for organic structures and their nuclear magnetic resonance (nmr) spectra.

1. [10016316](#)

Reactions & Pathways 18

BioModels 2
Database of Mathematical models of biological interest.

1. [BIOMD0000000241](#)
Shi1993_Caffeine_pressor_tolerance
2. [BIOMD0000000601](#)
Rosas2015 - Caffeine-induced luminal SR calcium changes

[show all >>](#)

BKMS-react 3
BKMS-react is an integrated and non-redundant biochemical reaction database containing known enzyme-catalyzed and spontaneous reactions.

1. [882](#)
2. [7965](#)
3. [51266](#)

[show all >>](#)

Rhea 6
Rhea is a freely available, manually annotated database of biochemical reactions.

Fig. 3.4 External references related to caffeine

In the Protein Sequences section, we have 77 proteins (in September of 2018) related to *caffeine*. If we click on *show all* we will get the complete list⁵ (see Fig. 3.5). These links are to another resource hosted by the EBI, the UniProt, a database of protein sequences and annotation data.

UniProt Automatically Generated Cross-References

Version 2014_02 of UniProt was used for these cross-references.

77 entries found, displaying 1 to 15.

Identifiers	Name	Line Types
A2AGL3	Ryanodine receptor 3	CC - MISCELLANEOUS
A4GE69	7-methylxanthosine synthase 1	CC - FUNCTION
A4GE70	3,7-dimethylxanthine N-methyltransferase	CC - CATALYTIC ACTIVITY; CC - FUNCTION
A6MFK9	Cysteine-rich venom protein	CC - FUNCTION
B0LPN4	Ryanodine receptor 2	CC - MISCELLANEOUS
B7FDI0	Cysteine-rich venom protein	CC - FUNCTION
B7FDI1	Cysteine-rich venom protein	CC - FUNCTION
B8QG00	Hadrucalcin	CC - FUNCTION
D7REY3	Caffeine dehydrogenase subunit alpha	DE; FT; CC - CATALYTIC ACTIVITY; CC - FUNCTION; CC - BIOPHYSICOCHEMICAL PROPERTIES
D7REY4	Caffeine dehydrogenase subunit beta	DE; FT; CC - CATALYTIC ACTIVITY; CC - FUNCTION; CC - BIOPHYSICOCHEMICAL PROPERTIES
D7REY5	Caffeine dehydrogenase subunit gamma	DE; FT; CC - CATALYTIC ACTIVITY; CC - FUNCTION; CC - BIOPHYSICOCHEMICAL PROPERTIES
E9PZQ0	Ryanodine receptor 1	CC - MISCELLANEOUS
E9Q401	Ryanodine receptor 2	CC - MISCELLANEOUS
F0E1K6	Probable methylxanthine N7-demethylase NdmC	CC - FUNCTION
F1LMY4	Ryanodine receptor 1	CC - MISCELLANEOUS

Export options: [CSV](#) | [Excel](#) | [XML](#)

Fig. 3.5 Proteins related to caffeine

The list includes the identifiers of each protein with a direct link to respective entry in UniProt, the name of the protein and some topics about the description of the protein. For example, DISRUPTION PHENOTYPE means some effects caused by the disruption of the gene coding for the protein are known⁶.

We should note that at bottom-right of the page there are *Export options* that enable us to download the full list of protein references in a single file. These options include:

CSV:

Comma Separated Values, the open format file that enable us to store data as a single table format (columns and rows).

Excel:

a proprietary format designed to store and access the data using the software Microsoft Excel.

XML:

eXtensible Markup Language, the open format file that enable us to store data using a hierarchy of markup tags.

We start by downloading the CSV, Excel and XML files. We can now open the files and check its contents in a regular text editor software⁷ installed in our computer, such as notepad (Windows), TextEdit (Mac) or gedit (Linux).

The first lines of the *chebi_27732_xrefs_UniProt.csv* file should look like this:

```
A2AGL3,Ryanodine receptor 3,CC -  
    MISCELLANEOUS  
A4GE69,7-methylxanthosine  
    synthase 1,CC - FUNCTION
```

The first lines of the *chebi_27732_xrefs_UniProt.xls* file should look like this:

```
"Identifiers" "Name"  
                "Line Types"  
"A2AGL3"      "Ryanodine  
    receptor 3"      "CC -  
    MISCELLANEOUS"  
"A4GE69"      "7-  
    methylxanthosine synthase 1"  
    "CC - FUNCTION"
```

As we can see, this is not the proprietary format XLS but instead a TSV format. Thus, the file can still be open directly on *Microsoft Excel*.

The first lines of the *chebi_27732_xrefs_UniProt.xml* file should look like this:

```
<?xml version="1.0"?>  
<table>  
<row>  
<column>A2AGL3</column>
```

```

<column>Ryanodine receptor 3</
column>
<column>CC - MISCELLANEOUS</
column>
</row>
<row>
<column>A4GE69</column>
<column>7-methylxanthosine
synthase 1</column>
<column>CC - FUNCTION</column>
</row>

```

...
We should note that all the files contain the same data they only use a different format.

If for any reason, we are not able to download the previous files from UniProt, we can get them from the book file archive⁸.

In the following sections we will use these files to automatize this process, but for now let us continue our manual exercise using the internet browser. Let us select the *Ryanodine receptor 1* with the identifier P21817 and click on the link⁹ (see Fig. 3.6). We can now see that UniProt is much more than just a sequence database. The sequence is just a tiny fraction of all the information describing the protein. All this information can also be downloaded as a single file by clicking on Format and on XML. Then, save the result as a XML file to our computer.

UniProtKB - P21817 (RYR1_HUMAN)

Display | BLAST | Align | Format | Add to basket | History | Feedback | Help video | Other tutorials and videos

Entry | Publications | Feature viewer | Feature table

Protein | **Ryanodine receptor 1**

Gene | **RYR1**

Organism | *Homo sapiens (Human)*

Status | Reviewed - Annotation score: ●●●●● - Experimental evidence at protein levelⁱ

Functionⁱ

Calcium channel that mediates the release of Ca^{2+} from the sarcoplasmic reticulum into the cytoplasm and thereby plays a key role in triggering muscle contraction following depolarization of T-tubules (PubMed:11741831, PubMed:16163667). Repeated very high-level exercise increases the open probability of the channel and leads to Ca^{2+} leaking into the cytoplasm (PubMed:18268335). Can also mediate the release of Ca^{2+} from intracellular stores in neurons, and may thereby promote prolonged Ca^{2+} signaling in the brain. Required for normal embryonic development of muscle fibers and skeletal muscle. Required for normal heart morphogenesis, skin development and ossification during embryogenesis (By similarity).

By similarity | 2 Publications | 1 Publication

Miscellaneous

Fig. 3.6 UniProt entry describing the Ryanodine receptor 1

Again, we can use our text editor to open the downloaded file named *P21817.xml*, which first lines should look like this:

```

<?xml version='1.0' encoding='
  UTF-8'?>
<uniprot xmlns="http://uniprot.
  org/uniprot" xmlns:xsi="http:
  //www.w3.org/2001/XMLSchema-
  instance" xsi:schemaLocation=
  "http://uniprot.org/uniprot
  http://www.uniprot.org/
  support/docs/uniprot.xsd">
<entry dataset="Swiss-Prot"
  created="1991-05-01" modified
  ="2018-06-20" version="210">
<accession>P21817</accession>

```

...
 We can check that this entry represents a *Homo sapiens* (Human) protein, so if we are interested only in Human Proteins, we will have to filter them. For example, the entry E9PZQ0¹⁰ in the ChEBI list also represents a *Ryanodine receptor 1* protein but for the *Mus musculus* (Mouse).

Going back to the browser in the top-left side of the UniProt entry we have a link to publications¹¹. If we click on it, we will see a list of publications somehow related to the protein (see Fig. 3.7).

UniProtKB

BLAST Align Retrieve/ID mapping Peptide search Help Contact

Publications related to P21817 - RYR1_HUMAN

1 to 25 of 237 Show 25

Display

Entry

Publications

Feature viewer

Feature table

Filter by

Source

UniProtKB/Swiss-Prot (78) (reviewed)

Computationally mapped (159)

Category

1. "Molecular cloning of cDNA encoding human and rabbit forms of the Ca²⁺ release channel (ryanodine receptor) of skeletal muscle sarcoplasmic reticulum."
Zorzato F., Fujii J., Otsu K., Phillips M.S., Green N.M., Lai F.A., Meissner G., MacLennan D.H.
J. Biol. Chem. 265:2244-2256(1990) [PubMed] [Europe PMC] [Abstract]
Cited for: NUCLEOTIDE SEQUENCE [MRNA] (ISOFORM 2), PARTIAL PROTEIN SEQUENCE.
Category: Sequences.
Tissue: Skeletal muscle.
Source: UniProtKB/Swiss-Prot (reviewed).
This publication is cited by 1 and mapped to 9 other entries.
2. "Polymorphisms and deduced amino acid substitutions in the coding sequence of the ryanodine receptor (RYR1) gene in individuals with malignant hyperthermia."
Gillard E.F., Otsu K., Fujii J., Duff C.L., de Leon S., Khanna V.K., Britt B.A., Worton R.G., McLennan D.H.
Genomics 13:1247-1254(1992) [PubMed] [Europe PMC] [Abstract]
Cited for: SEQUENCE REVISION TO 2324; 2840 AND 3380, INVOLVEMENT IN MHS1, VARIANT MHS1 ARG-248, VARIANTS CYS-471; LEU-1787; CYS-2060 AND VAL-2550.
Category: Pathology & Biotech, Sequences.
Tissue: Muscle.
Source: UniProtKB/Swiss-Prot (reviewed).
3. "A mutation in the human ryanodine receptor gene associated with central core disease."

Fig. 3.7 Publications related to Ryanodine receptor 1

Let us assume that we are interested in finding phenotypic information, the first title that may attract our attention is: *Polymorphisms and deduced amino acid substitutions in the coding sequence of the ryanodine receptor (RYR1) gene in individuals with malignant hyperthermia*. To know more about the publication, we can use the UniProt citations service by clicking on the Abstract link¹² (see Fig. 3.8).

Polymorphisms and deduced amino acid substitutions in the coding sequence of the ryanodine receptor (RYR1) gene in individuals with malignant hyperthermia.

Gillard E.F., Otsu K., Fujii J., Duff C.L., de Leon S., Khanna V.K., Britt B.A., Worton R.G., McLennan D.H.

Twenty-one polymorphic sequence variants of the RYR1 gene, including 13 restriction fragment length polymorphisms (RFLPs), were identified by sequence analysis of human ryanodine receptor (RYR1) cDNAs from three individuals predisposed to malignant hyperthermia (MH). All RFLPs were detectable in PCR-amplified products, and their segregation was consistent with our initial finding of linkage to MH in the nine families previously informative for one or more intragenic markers (MacLennan et al., 1990, Nature 343:559-561). Four amino acid substitutions were identified in the study: Arg for Gly248, Cys for Arg470, Leu for Pro1785, and Cys for Gly2059. Of 45 families tested, a single family presented the Arg for Gly248 substitution where it segregated with malignant hyperthermia, making it a candidate mutation for predisposition to MH in man. The other three polymorphic substitutions failed to segregate with malignant hyperthermia in those families in which they occurred, implying that they represent polymorphisms with little or no effect on the function of the RYR1 gene.

Genomics 13:1247-1254(1992) [PubMed] [Europe PMC]

Fig. 3.8 Abstract of the publication entitled *Polymorphisms and deduced amino acid substitutions in the coding sequence of the ryanodine receptor (RYR1) gene in individuals with malignant hyperthermia*

To check if the abstract mentions any disease we can use an online text mining tool, for example the Minimal Named-Entity Recognizer (MER)¹³. We can copy and paste the abstract of the publication into MER and select *DO – Human Disease Ontology* as lexicon (see Fig. 3.9).

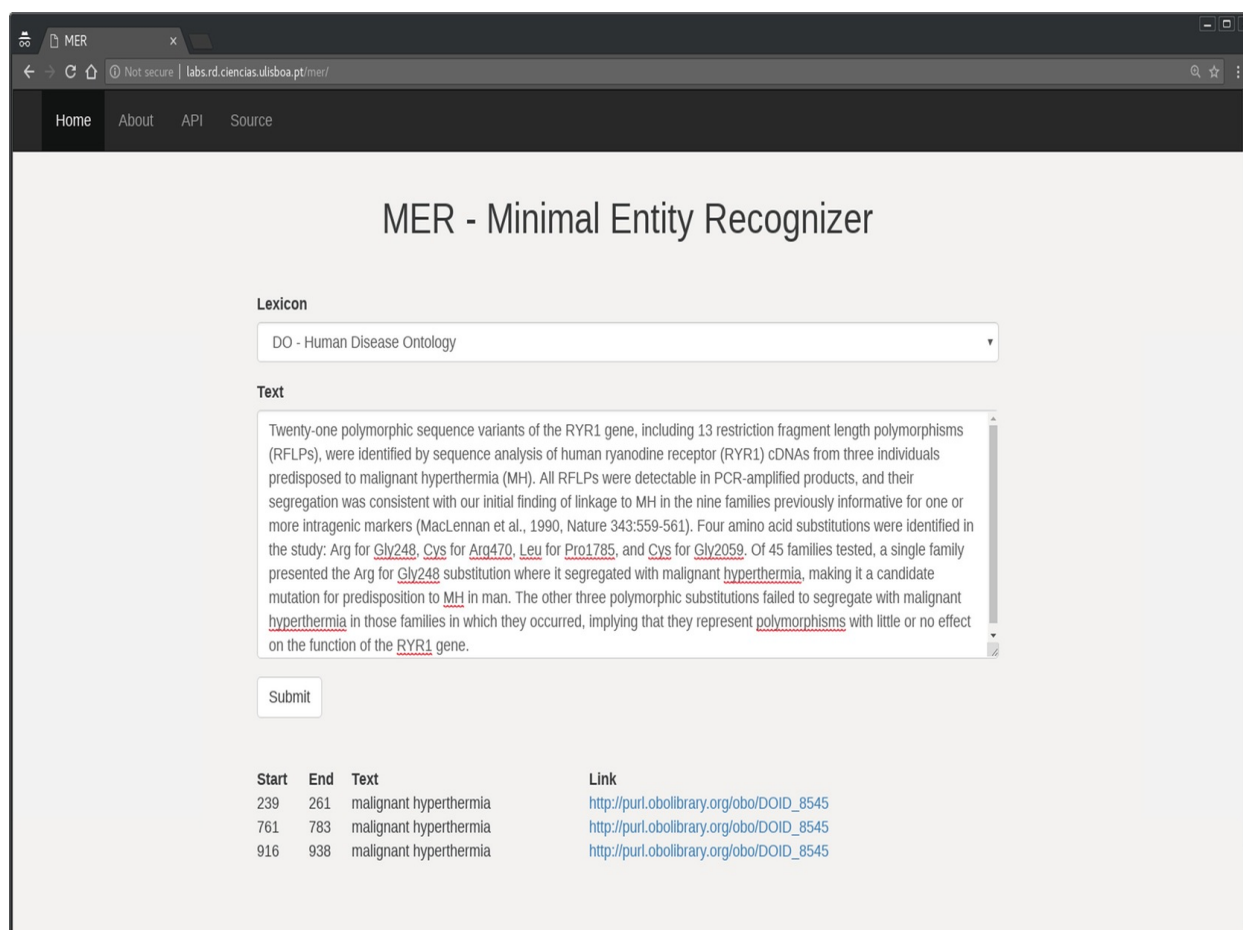


Fig. 3.9 Diseases recognized by the online tool MER in an abstract

We will see that MER detects three mentions of *malignant hyperthermia*, giving us another link¹⁴ about the disease found (see Fig. 3.10).

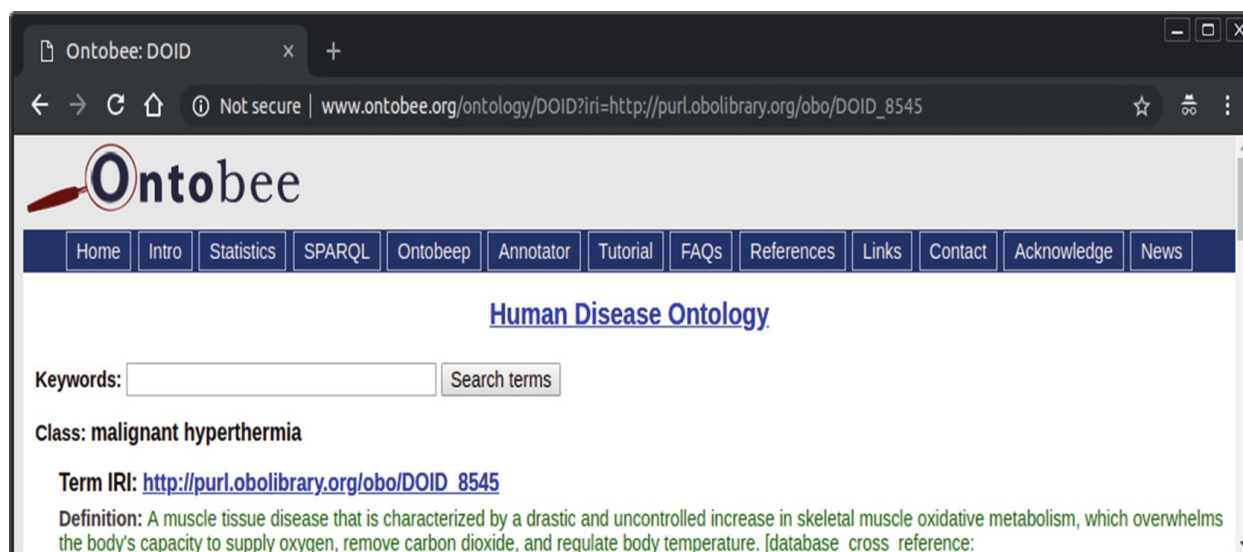


Fig. 3.10 Ontobee entry for the class *malignant hyperthermia*

Thus, in summary, we started from a generic definition of *caffeine* and ended with an abstract about hyperthermia by following the links in different databases. Of course, this does not mean that by taking *caffeine* we will get hyperthermia, or that we will treat hyperthermia by taking *caffeine* (maybe as a cold drink 😊¹⁵). However, this relation has a context, a protein and a publication, that need to be further analyzed before drawing any conclusions.

We should note that we only analyzed one protein and one publication, we now need to repeat all the steps to all the proteins and to all the publications related to each protein. And this could even be more complicated if we were interested in other central nervous system stimulants, for example by looking in the ChEBI ontology¹⁶. This is of course the motivation to automatize the process, since it is not humanly feasible to deal with such large amount of data, that keeps evolving every day.

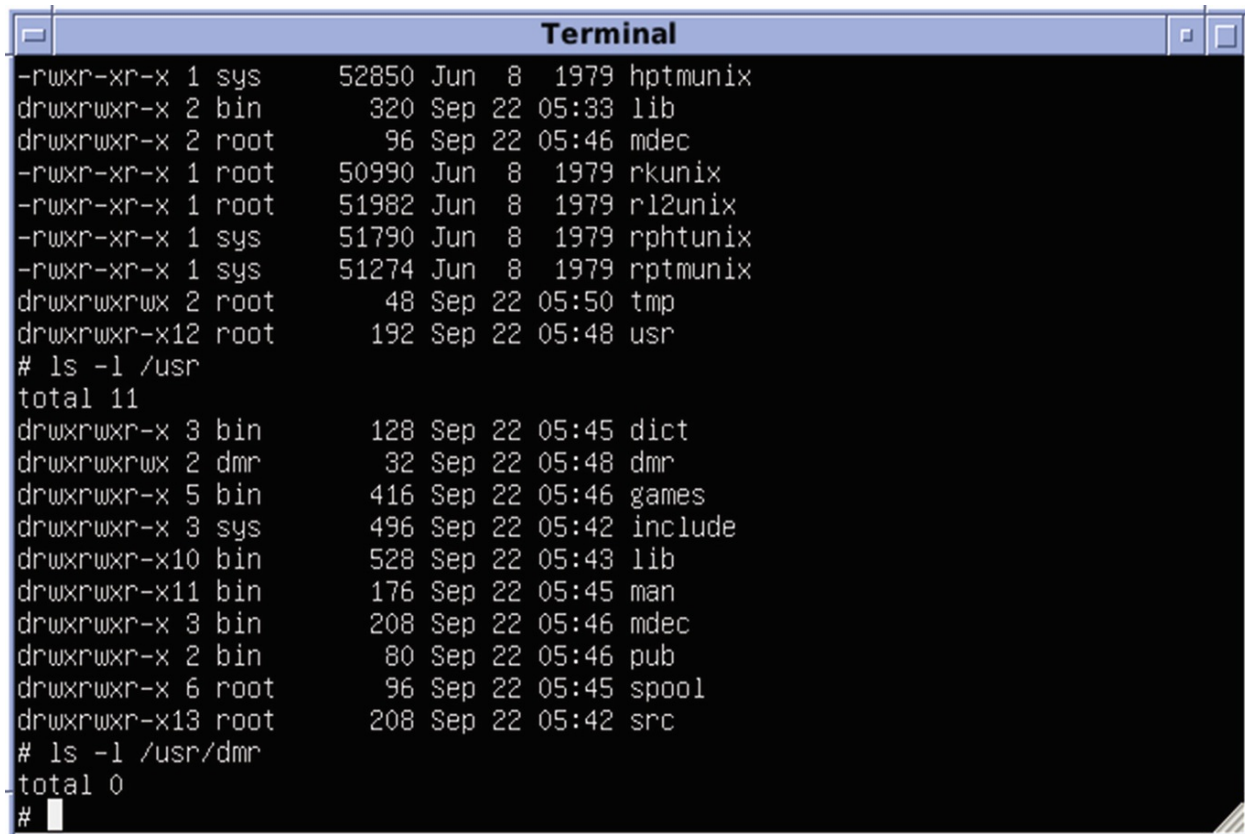
However, if the goal was to find a relation between *caffeine* and hyperthermia, we could simply have searched these two terms in PubMed. We did not do that because some relations are not explicitly mention in the text, thus we have to navigate through database links. The second reason is because we needed an example using different resources and multiple entries to explain how we can automate most of these steps using shell scripting. The automation of the example will introduce a comprehensive set of techniques and commands, which with some adaptation Life and Health specialists can use to address many of their text and data processing challenges.

Unix Shell

The first step is to open a shell in our personal computer. A shell is a software program that interprets and executes command lines given by the user in consecutive lines of text. A shell script is a list of such command lines. The command line usually starts by invoking a command line tool. This manuscript will introduce a few command line tools, which will allow us to automatize the previous example. Unix shell was developed to manage Unix-like operating systems, but due to their usefulness nowadays they are available in most personal computers using Linux, macOS or Windows operating systems. There are many types of Unix shells with minor differences between them (e.g. sh, ksh, csh, tcsh and bash), but the most widely available is the Bourne-Again shell (bash¹⁷). The examples in this manuscript were tested using bash.

So, the first step is to open a shell in our personal computer using a terminal

application (see Fig. 3.11). If we are using Linux or macOS then this is usually not new for us, since most probably we have a terminal application already installed, that opens a shell for us. In case we are using a Microsoft Windows operating system, then we have several options to consider. If we are using Windows 10, then we can install a Windows Subsystem for Linux¹⁸ or just install a third-party application, such as MobaXterm¹⁹. No matter which terminal application we end up using, the shell will always have a common look: a text window with a cursor blinking waiting for our first command line. We should note that most terminal applications allow the usage of the up and down cursor keys to select, edit, and execute previous commands, and the usage of the tab key to complete the name of a command or a file.



```
Terminal
-rwxr-xr-x 1 sys 52850 Jun 8 1979 hptmunix
drwxrwxr-x 2 bin 320 Sep 22 05:33 lib
drwxrwxr-x 2 root 96 Sep 22 05:46 mdec
-rwxr-xr-x 1 root 50990 Jun 8 1979 rkunix
-rwxr-xr-x 1 root 51982 Jun 8 1979 rl2unix
-rwxr-xr-x 1 sys 51790 Jun 8 1979 rphtunix
-rwxr-xr-x 1 sys 51274 Jun 8 1979 rptmunix
drwxrwxrwx 2 root 48 Sep 22 05:50 tmp
drwxrwxr-x12 root 192 Sep 22 05:48 usr
# ls -l /usr
total 11
drwxrwxr-x 3 bin 128 Sep 22 05:45 dict
drwxrwxrwx 2 dmr 32 Sep 22 05:48 dmr
drwxrwxr-x 5 bin 416 Sep 22 05:46 games
drwxrwxr-x 3 sys 496 Sep 22 05:42 include
drwxrwxr-x10 bin 528 Sep 22 05:43 lib
drwxrwxr-x11 bin 176 Sep 22 05:45 man
drwxrwxr-x 3 bin 208 Sep 22 05:46 mdec
drwxrwxr-x 2 bin 80 Sep 22 05:46 pub
drwxrwxr-x 6 root 96 Sep 22 05:45 spool
drwxrwxr-x13 root 208 Sep 22 05:42 src
# ls -l /usr/dmr
total 0
#
```

Fig. 3.11 Screenshot of a Terminal application (Source: <https://en.wikipedia.org/wiki/Unix>)

Current Directory

As our first command line, we can type:

```
$ pwd
```

After hitting enter, the command will show the full path of the directory (folder) of our computer in which the shell is working on. The dollar sign in the

left is only to indicate that this is a command to be executed directly in the shell.

To understand a command line tool, such as `pwd`, we can type `man` followed by the name of the tool. For example, we can type `man pwd` to learn more about `pwd` (do not forget to hit enter, and press `q` to quit). We can also learn more about `man` by typing `man man`. A shorter alternative to `man`, is to add the `--help` option after any command tool. For example, we can type `pwd --help` to have a more concise description of `pwd`.

As our second command line, we can type `ls` and hit enter. It will show the list of files in the current directory. For example, we can type `ls --help` to have a concise description of `ls`. Since we will work with files, that we need to open with a text editor or a spreadsheet application²⁰, such as LibreOffice Calc or Microsoft Excel, we should select a current directory that we can easily open in our file explorer application. A good idea is to open our favorite file explorer application, select a directory, and then check its full path²¹.

Windows Directories

Notice that in Windows the full path to a directory each name is separated by a backslash (`\\`) while in a Unix shell is a forward slash (`/`).

For example, a Windows path to the Documents folder may look like:

```
C:\Users\MyUserName\Documents
```

If we are using the Windows Subsystem for Linux²², the previous folder must be accessed using the path:

```
/mnt/c/Users/MyUserName/  
Documents
```

If we are using MobaXterm²³, the following path should be used instead:

```
/drives/c/Users/MyUserName/  
Documents
```

Change Directory

To change the directory, we can use another command line tool, the `cd` (change directory) followed by the new path. In a Linux system we may want to use the *Documents* directory. If the *Documents* directory is inside our current directory (shown using `ls`), we only need to type:

```
$ cd Documents
```

Now we can type `pwd` to see what changed.

And if we want to return to the parent directory, we only need to use the two dots `..`:

```
$ cd ..
```

And if we want to return to the home directory, we only need to use the tilde

character (~):

```
$ cd ~
```

Again, we should type `pwd` to double check if we are in the directory we really want.

In Windows we may need to use the full path, for example:

```
$ cd /mnt/c/Users/MyUserName/  
Documents
```

We should note that we need to enclose the path within single (or double) quotes in case it contains spaces:

```
$ cd '/mnt/c/Users/MyUserName/  
Documents'
```

Later on, we will know more about the difference between using single or double quotes. For now, we may assume that they are equivalent. To know more about `cd`, we can type `cd --help`.

Useful Key Combinations

Every time the terminal is blocked by any reason, we can press both the control and C key at the same time²⁴. This usually cancels the current tool being executed. For example, try using the `cd` command with only one single quote:

```
$ cd '
```

This will block the terminal, because it is still waiting for a second single quote that closes the argument. Now press control-C, and the command will be aborted.

Now we can type again the previous command, but instead of pressing control-C we may also press control-D²⁵. The combination control-D indicates the terminal that it is the end of input. So, in this case, the `cd` command will not be canceled, but instead it is executed without the second single quote and therefore a syntax error will be shown on our display.

Other useful key combinations are the control-L that when pressed cleans the terminal display, and the control-insert and shift-insert that when pressed copy and paste the selected text, respectively.

Shell Version

The following examples will probably work in any Unix shell, but if we want to be certain that we are using bash we can type the following command, and check if the output says bash.

```
$ ps -p $$
```

`ps` is a command line tool that shows information about active processes running in our computer. The `-p` option selects a given process, and in this case

\\$ \\$ represents the process running in our terminal application. In most terminal applications bash is the default shell. If this is not our case, we may need to type `bash`, hit enter and now we are using bash.

Now that we know how to use a shell, we can start writing and running a very simple script that reverse the order of the lines in a text file.

Data File

We start by creating a file named *myfile.txt* using any text editor, and adding the following lines:

```
line 1
line 2
line 3
line 4
```

We cannot forget to save it in our working directory, and check if it has the proper filename extension.

File Contents

To check if the file is really on our working directory, we can type:

```
$ cat myfile.txt
```

The contents of the file should appear in our terminal. `cat` is a simple command line tool that receives a filename as argument and displays its contents on the screen. We can type `man cat` or `cat --help` to know more about this command line tool.

Reverse File Contents

An alternative to `cat` tool is the `tac` tool. To try it, we only need to type:

```
$ tac myfile.txt
```

The contents of the file should also appear in our terminal, but now in the reverse order. We can type `man tac` or `tac --help` to know more about this command line tool.

My First Script

Now we can create a script file named *reversemyfile.sh* by using the text editor, and add the following lines:

```
1 tac $1
```

We cannot forget to save the file in our working directory. `\$1` represents the first argument after the script filename when invoking it. Each script file presented in this manuscript will include the line numbers in the left. This will helps us not only to identify how many lines the script contains, but also to

distinguish a script file from the commands to be executed directly in the shell.

Line Breaks

A Unix file represents a single line break by a line feed character, instead of two characters (carriage return and line feed) used by Windows²⁶. So, if we are using a text editor in Windows, we must be careful to use one that lets us save it as Unix file, for example the open source Notepad++²⁷.

In case we do not have such text editor, we can also remove the extra carriage return by using the command line tool `tr`, that replaces and deletes characters:

```
$ tr -d '\r' < reversemyfile.sh
> reversemyfilenew.sh
```

The `-d` option of `tr` is used to remove a given character from the input, in this case `tr` will delete all carriage returns (`\r`). Many command line options can be used in short form using a single dash (`-`), or in a long form using two dashes (`--`). In this tool, using the `--delete` option is equivalent to the `-d` option. Long forms are more self-explanatory, but they take longer to type and occupy more space. We can type `man tr` or `tr --help` to know more about this command line tool.

Redirection Operator

The `>` character represents a redirection operator²⁸ that moves the results being displayed at the standard output (our terminal) to a given file. The `<` character represents a redirection operator that works on the opposite direction, i.e. opens a given file and uses it as the standard input.

We should note that `cat` received the filename as an input argument, while `tr` can only receive the contents of the file through the standard input. Instead of providing the filename as argument, the `cat` command can also receive the contents of a file through the standard input, and produce the same output:

```
$ cat < myfile.txt
```

The previous `tr` command used a new file for the standard output, because we cannot use the same file to read and write at the same time. To keep the same filename, we have to move the new file by using the `mv` command:

```
$ mv reversemyfilenew.sh
reversemyfile.sh
```

We can type `man mv` or `mv --help` to know more about this command line tool.

Installing Tools

These two last commands could be replaced by the `dos2unix` tool:

```
$ dos2unix -n reversemyfile.sh
```

If not available, we have to install the `dos2unix` tool. For example, in the Ubuntu Windows Subsystem we need to execute:

```
$ apt install dos2unix
```

The `apt` (Advanced Package Tool) command is used to install packages in many Linux systems²⁹. Another popular alternative is the `yum` (Yellowdog Updater, Modified) command³⁰.

To avoid fixing line breaks each time we update our file when using Windows, a clearly better solution is to use a Unix friendly text editor.

When we are not using Windows, or we are using a Unix friendly text editor, the previous commands will execute but nothing will happen to the contents of *reversemyfile.sh*, since the `tr` command will not remove any character. To see the command working replace '`\\r`' by '`\$`' and check what happens.

Permissions

A script also needs permission to be executed, so every time we create a new script file we need to type:

```
$ chmod u+x reversemyfile.sh
```

The command line tool `chmod` just gave the user (`u`) permissions to execute (`+x`). We can type `man chmod` or `chmod --help` to know more about this command line tool.

Finally, we can execute the script by providing the *myfile.txt* as argument:

```
$ ./reversemyfile.sh myfile.txt
```

The contents of the file should appear in our terminal in the reverse order:

```
line 4
```

```
line 3
```

```
line 2
```

```
line 1
```

Congratulations, we made our first script work ! 😊

If we give more arguments, they will be ignored:

```
$ ./reversemyfile.sh myfile.txt  
myotherfile.txt 'my other  
file.txt'
```

The output will be exactly the same because our script does not use `\$2` and `\$3`, that in this case will represent *myotherfile.txt* and *my other file.txt*, respectively. We should note that when containing spaces, the argument must be

enclosed by single quotes.

Debug

If something is not working well, we can debug the entire script by typing:

```
$ bash -x reversemyfile.sh  
    myfile.txt
```

Our terminal will not only display the resulting text, but also the command line tools executed preceded by the plus character (+):

```
+ tac myfile.txt  
line 4  
line 3  
line 2  
line 1
```

Alternatively, we can add the `set -x` command line in our script to start the debugging mode, and `set +x` to stop it.

Save Output

We can now save the output into another file named *mynewfile.txt* by typing:

```
$ ./reversemyfile.sh myfile.txt  
    > mynewfile.txt
```

Again, to check if the file was really created, we can use the `cat` tool:

```
$ cat mynewfile.txt
```

Or, we can reverse it again by typing:

```
$ ./reversemyfile.sh mynewfile.  
    txt
```

Of course, the result should exactly be the original contents of *myfile.txt*.

Web Identifiers

The input argument(s) of our retrieval task is the chemical compound(s) of which we want to retrieve more information. For the sake of simplicity, we will start by assuming that the user knows the ChEBI identifier(s), i.e. the script does not have to search by the name of the compounds. Nevertheless, to find the identifier of a compound by its name is also possible, and this manuscript will describe how to do it later on.

So, the first step, is to automatically retrieve all proteins associated to the given input chemical compound, that in our example was *caffeine* (CHEBI:27732). In the manual process, we downloaded the files by manually clicking on the links shown as *Export options*, namely the URLs:

```
https://www.ebi.ac.uk/chebi/  
viewDbAutoXrefs.do?d-1169080-  
e=1&6578706f7274=1&chebiId  
=27732&dbName=UniProt
```

```
https://www.ebi.ac.uk/chebi/  
viewDbAutoXrefs.do?d-1169080-  
e=2&6578706f7274=1&chebiId  
=27732&dbName=UniProt
```

```
https://www.ebi.ac.uk/chebi/  
viewDbAutoXrefs.do?d-1169080-  
e=3&6578706f7274=1&chebiId  
=27732&dbName=UniProt
```

for downloading a CSV, Excel, or XML file, respectively.

We should note that the only difference between the three URLs is a single numerical digit (1, 2, and 3) after the first equals character (=), which means that this digit can be used as an argument to select the type of file. Another parameter that is easily observable is the ChEBI identifier (27732). Try to replace 27732 by 17245 in any of those URLs by using a text editor, for example:

```
https://www.ebi.ac.uk/chebi/  
viewDbAutoXrefs.do?d-1169080-  
e=1&6578706f7274=1&chebiId  
=17245&dbName=UniProt
```

Now we can use this new URL in the internet browser, and check what happens. If we did it correctly, our browser downloaded a file with more than seven hundred proteins, since the 17245 is the ChEBI identifier of a popular chemical compound in life systems, the *carbon monoxide*.

In this case, we are not using a fully RESTful web service, but the data path is pretty modular and self-explanatory. The path is clearly composed of:

- the name of the database (chebi);
- the method (viewDbAutoXrefs.do);
- and a list of parameters and their value (arguments) after the question mark character (?).

The order of the parameters in the URL is normally not relevant. They are separated by the ampersand character (&) and the equals character (=) is used to assign a value to each parameter (argument). This modular structure of these URLs allows us to use them as data pipelines to fill our local files with data, like pipelines that transport oil or gas from one container to another.

Single and Double Quotes

To construct the URL for a given ChEBI identifier, let us first understand the difference between single quotes and double quotes in a string (sequence of characters). We can create a script file named *getproteins.sh* by using a text editor to add the following lines:

```
1 echo 'The input: $1'
2 echo "The input: $1"
```

The command line tool `echo` displays the string received as argument. Do not forget to save it in our working directory and add the right permissions with `chmod` as we did previously with our first script.

Now to execute the script we will only need to type:

```
$ ./getproteins.sh
```

The output on the terminal should be:

```
The input: $1
```

```
The input:
```

This means that when using single quotes, the string is interpreted literally as it is, whereas the string within double quotes is analyzed, and if there is a special character, such as the dollar sign (`\$`), the script translates it to what it represents. In this case, `\$1` represents the first input argument. Since no argument was given, the double quotes displays nothing.

To execute the script with an argument, we can type:

```
$ ./getproteins.sh 27732
```

The output on our terminal should be:

```
The input: $1
```

```
The input: 27732
```

We can check now that when using double quotes `\$1` is translated to the string given as argument.

Now we can update our script file named *getproteins.sh* to contain only the following line:

```
1 echo "https://www.ebi.ac.uk/
    chebi/viewDbAutoXrefs.do?d
    -1169080-e=1&6578706f7274
    =1&chebiId=$1&dbName=
    UniProt"
```

Comments

Instead of removing the previous lines, we can transform them in comments by adding the hash character (`\ #`) to the beginning of the line:

```

1 #echo 'The input: $1'
2 #echo "The input: $1"
3 echo "https://www.ebi.ac.uk/
    chebi/viewDbAutoXrefs.do?d
    -1169080-e=1&6578706f7274
    =1&chebiId=$1&dbName=
    UniProt"

```

Commented lines are ignored by the computer when executing the script.

Now, we can execute the script giving the ChEBI identifier as argument:

```
$ ./getproteins.sh 27732
```

The output on our terminal should be the link that returns the CSV file containing the proteins associated with *caffeine*.

Data Retrieval

After having the link, we need a web retrieval tool that works like our internet browser, i.e. receives as input a URL for programmatic access and retrieves its contents from the internet. We will use Client Uniform Resource Locator (cURL), which is available as a command line tool, and allows us to download the result of opening a URL directly into a file (man curl or curl --help for more information).

For example, to display in our screen the list of proteins related to *caffeine*, we just need to add the respective URL as input argument:

```
$ curl 'https://www.ebi.ac.uk/
    chebi/viewDbAutoXrefs.do?d
    -1169080-e=1&6578706f7274
    =1&chebiId=27732&dbName=
    UniProt'

```

In some systems the curl command needs to be installed³¹. Since we are using a secure connection *https*, we may also need to install the *ca-certificates* package³².

An alternative to curl is the command *wget*, which also receives a URL as argument but by default wget writes the contents to a file instead of displaying it on the screen (man wget or wget --help for more information). So, the equivalent command, is to add the *-O-* option to select where the contents is placed:

```
$ wget -O- 'https://www.ebi.ac.uk/chebi/viewDbAutoXrefs.do?d-1169080-e=1&6578706f7274=1&chebiId=27732&dbName=UniProt'
```

We should note that dash - character after -O represents the standard output. The equivalent long form to the -O option is --output-document=file. The output on our terminal should be the long list of proteins:

```
Q15413,Ryanodine receptor 3,CC -
MISCELLANEOUS
Q92375,Thioredoxin reductase,DE
Q92736,Ryanodine receptor 2,CC -
MISCELLANEOUS
```

Instead of using a fixed URL, we can update the script named *getproteins.sh* to contain only the following line:

```
1 curl "https://www.ebi.ac.uk/chebi/viewDbAutoXrefs.do?d-1169080-e=1&6578706f7274=1&chebiId=$1&dbName=UniProt"
```

We should note that now we are using double quotes, since we replaced the *caffeine* identifier by `\$1`.

Now to execute the script we only need to provide a ChEBI identifier as input argument:

```
$ ./getproteins.sh 27732
```

The output on our terminal should be the long list of proteins:

```
Q15413,Ryanodine receptor 3,CC -
MISCELLANEOUS
Q92375,Thioredoxin reductase,DE
Q92736,Ryanodine receptor 2,CC -
MISCELLANEOUS
```

Or, if we want the proteins related to *carbon monoxide*, we only need to replace the argument:

```
$ ./getproteins.sh 17245
```

And the output on our terminal should be an even longer list of proteins:

```

...
Q58432,Phosphomethylpyrimidine
    synthase,CC - CATALYTIC
    ACTIVITY
Q62976,Calcium-activated
    potassium channel subunit
    alpha-1,CC - ENZYME
    REGULATION; CC - DOMAIN
Q63185,Eukaryotic translation
    initiation factor 2-alpha
    kinase 1,CC - ENZYME
    REGULATION

```

If we want to analyze all the lines we can redirect the output to the command line tool `less`, which allows us to navigate through the output by using the arrow keys. To do that we can add the bar character (`|`) between two commands, which will transfer the output of the first command as input of the second:

```
$ ./getproteins.sh 27732 | less
```

To exit from `less` just press `q`.

However, what we really want is to save the output as a file, not just printing some characters on the screen. Thus, what we should do is redirect the output to a CSV file. This can be done by adding the redirect operator `>` and the filename, as described previously:

```
$ ./getproteins.sh 27732 >
    chebi_27732_xrefs_UniProt.
    csv

```

We should note that `curl` still prints some progress information into the terminal.

Standard Error Output

This happens because it is displaying that information into the standard error output, which was not redirected to the file³³. The `>` character without any preceding number by default redirects the standard output. The same happens if we precede it by the number 1. If we do not want to see that information, we can also redirect the standard error output (2), but in this case to the null device (`/dev/null`):

```
$ ./getproteins.sh 27732 >
    chebi_27732_xrefs_UniProt.
    csv 2>/dev/null

```

We can also use the `-s` option of `curl` in order to suppress the progress information, by adding it to our script file named `getproteins.sh`:

```
1 curl -s "https://www.ebi.ac.uk
    /chebi/viewDbAutoXrefs.do?
    d-1169080-e=1&6578706f7274
    =1&chebiId=$1&dbName=
    UniProt"
```

The equivalent long form to the `-s` option is `--silent`.

Now when executing the script, no progress information is shown:

```
$ ./getproteins.sh 27732 >
    chebi_27732_xrefs_UniProt.
    csv
```

To check if the file was really created and to analyze its contents, we can use the `less` command:

```
$ less chebi_27732_xrefs_UniProt
    .csv
```

We can also open the file in our spreadsheet application, such as LibreOffice Calc or Microsoft Excel.

As an exercise execute the script to get the CSV file with the associated proteins of water³⁴ and gold³⁵.

Data Extraction

Some data in the CSV file may not be relevant regarding our information need, i.e. we may need to identify and extract relevant data. In our case, we will select the relevant proteins (lines) using the command line tool `grep`, and secondly, we will select the column we need using the command line tool `gawk`, which is the GNU implementation of `awk`³⁶. We should note that if we are using MobaXterm we may need to install the `gawk` package³⁷. We can also replace `gawk` by `awk` in case another implementation is available³⁸.

Since our information need is about diseases related to *caffeine*, we may assume that we are only interested in proteins that have one of these topics in the third column:

```
CC - MISCELLANEOUS
CC - DISRUPTION PHENOTYPE
CC - DISEASE
```

Extracting lines from a text file is the main function of `grep`. The selection is performed by giving as input a pattern that `grep` tries to find in each line,

presenting only the ones where it was able to find a match. The pattern is the same as the one we normally use when searching for a word in our text editor. The `grep` command also works with more complex patterns such as regular expressions, that we will describe later on.

Single and Multiple Patterns

We can execute the following command that selects the proteins with the topic CC - MISCELLANEOUS, our pattern, in our CSV file:

```
$ grep 'CC - MISCELLANEOUS'
    chebi_27732_xrefs_UniProt.
    csv
```

The output will be a shorter list of proteins, all with CC - MISCELLANEOUS as topic:

```
A2AGL3,Ryanodine receptor 3,CC -
    MISCELLANEOUS
B0LPN4,Ryanodine receptor 2,CC -
    MISCELLANEOUS
E9PZQ0,Ryanodine receptor 1,CC -
    MISCELLANEOUS
E9Q401,Ryanodine receptor 2,CC -
    MISCELLANEOUS
F1LMY4,Ryanodine receptor 1,CC -
    MISCELLANEOUS
P11716,Ryanodine receptor 1,CC -
    MISCELLANEOUS
P21817,Ryanodine receptor 1,CC -
    DISEASE; CC - MISCELLANEOUS
P54867,Protein SLG1,CC -
    MISCELLANEOUS
Q9TS33,Ryanodine receptor 3,CC -
    MISCELLANEOUS
Q15413,Ryanodine receptor 3,CC -
    MISCELLANEOUS
Q92736,Ryanodine receptor 2,CC -
    MISCELLANEOUS
```

To use multiple patterns, we must precede each pattern with the `-e` option:

```
$ grep -e 'CC - MISCELLANEOUS' -
    e 'CC - DISRUPTION
    PHENOTYPE' -e 'CC -
    DISEASE'
    chebi_27732_xrefs_UniProt.
    csv
```

The equivalent long form to the `-e` option is `--regexp=PATTERN`.

The output on our terminal should be a longer list of proteins:

```
Q9VSH2,Gustatory receptor for
    bitter taste 66a,CC -
    FUNCTION; CC - DISRUPTION
    PHENOTYPE
Q15413,Ryanodine receptor 3,CC -
    MISCELLANEOUS
Q92736,Ryanodine receptor 2,CC -
    MISCELLANEOUS
```

We should note that as previously, we can add `| less` to check all of them more carefully. The `less` command also gives the opportunity to find lines based on a pattern. We only need to type `/` and then a pattern.

We can now update our script file named *getproteins.sh* to contain the following lines:

```
1 curl -s "https://www.ebi.ac.uk
    /chebi/viewDbAutoXrefs.do?
    d=1169080-e=1&6578706f7274
    =1&chebiId=$1&dbName=
    UniProt" | \
2 grep -e 'CC - MISCELLANEOUS' -
    e 'CC - DISRUPTION
    PHENOTYPE' -e 'CC -
    DISEASE'
```

We should note that we added the `-s` option to suppress the progress information of `curl`, and the characters `| \` to the end of line to redirect the output of that line as input of the next line, in this case the `grep` command. We need to be careful in ensuring that `\` is the last character in the line, i.e. spaces in the end of the line may cause problems.

We can now execute the script again:

```
$ ./getproteins.sh 27732
```

The output should be similar of what we got previously, but the script downloads the data and filters immediately.

To save the file with the relevant proteins, we only need to add the redirection operator:

```
$ ./getproteins.sh 27732 >
    chebi_27732_xrefs_UniProt
    _relevant.csv
```

Data Elements Selection

Now we need to select just the first column, the one that contains the protein identifiers. Selecting columns from a tabular file is one easy task for `gawk`, that besides performing pattern scanning also provides a complex processing language (AWK³⁹). This processing language can be highly complex⁴⁰ and it is out of our scope for this introductory manuscript. The `gawk` command can receive as arguments the character that divides each data element (column) in a line using the `-F` option, and an instruction of what to do with it enclosed by single quotes and curly brackets. The equivalent long form to the `-F` option is `-field-separator=fs`.

For example, we can get the first column of our CSV file:

```
$ gawk -F, '{ print $1 }' <
    chebi_27732_xrefs_UniProt_
    relevant.csv
```

We should note that comma (,) is the character that separates data elements in a CSV file, and that `print` is equivalent to `echo`, and `\$1` represents the first data element.

The command will display only the first column of the file, i.e. the protein identifiers:

```
...
Q9VSH2
Q15413
Q92736
```

For example, we can get the first and third columns separated by a comma:

```
$ gawk -F, '{ print $1 ", " $3 }'
    < chebi_27732_xrefs_
    UniProt_relevant.csv
```

Now, the output contains both the first and third column of the file:

```
...
Q9VSH2, CC - FUNCTION; CC -
    DISRUPTION PHENOTYPE
Q15413, CC - MISCELLANEOUS
Q92736, CC - MISCELLANEOUS
```

We can update our script file named *getproteins.sh* to contain the following lines:

```
1 curl -s "https://www.ebi.ac.uk
    /chebi/viewDbAutoXrefs.do?
    d-1169080-e=1&6578706f7274
    =1&chebiId=$1&dbName=
    UniProt" | \
2 grep -e 'CC - MISCELLANEOUS' -
    e 'CC - DISRUPTION
    PHENOTYPE' -e 'CC -
    DISEASE' | \
3 gawk -F, '{ print $1 }'
```

The last line is the only that changes, except the `| \` in the previous line to redirect the output.

To execute the script, we can type again:

```
$ ./getproteins.sh 27732
```

The output should be similar of what we got previously, but now only the protein identifiers are displayed.

To save the output as a file with the relevant proteins' identifiers, we only need to add the redirection operator:

```
$ ./getproteins.sh 27732 >
    chebi_27732_xrefs_UniProt_
    relevant_identifiers.csv
```

Task Repetition

Given a protein identifier we can construct the URL that will enable us to download its information from UniProt. We can use the RESTful web services provided by UniProt⁴¹, more specifically the one that allow us to retrieve a specific entry⁴². The construction of the URL is simple, it starts always by `https://www.uniprot.org/uniprot/`, followed by the protein identifier, ending with a dot and the data format. For example, the link for protein *P21817* using the XML format is: <http://www.uniprot.org/uniprot/>

Assembly Line

However, we need to construct one URL for each protein from the list we previously retrieved. The size of the list can be large (hundreds of proteins), varies for different compounds and evolves with time. Thus, we need an assembly line in which a list of proteins identifiers, independently of its size, are added as input to commands that construct one URL for each protein and retrieve the respective file.

The `xargs` command line tool works as an assembly line, it executes a command per each line given as input. We should note that if we are using MobaXterm we may need to install the *findutils* package⁴³, since the default `xargs` only has minimal options⁴⁴.

We can start by experimenting the `xargs` command by giving as input the list of protein identifiers in file *chebi_27732_xrefs_UniProt_relevant_identifiers.csv*, display each identifier on the screen in the middle of a text message by providing the `echo` command as argument:

```
$ cat chebi_27732_xrefs_UniProt_
    relevant_identifiers.csv
    | xargs -I {} echo '
    Another protein id {} to
    retrieve'
```

The `xargs` command received as input the contents our CSV file, and for each line displayed a message including the identifier in that line. The `-I` option tells `xargs` to replace `\{\}` in the command line given as argument by the value of the line being processed. The equivalent long form to the `-I` option is `-replace=R`.

The output should be something like this:

```
Another protein id A2AGL3 to
    retrieve
Another protein id B0LPN4 to
    retrieve
Another protein id E9PZQ0 to
    retrieve
```

...

Instead of creating inconsequential text messages, we can use `xargs` to create the URLs:

```
$ cat chebi_27732_xrefs_UniProt
    _relevant_identifiers.csv
    | xargs -I {} echo 'https
      ://www.uniprot.org/uniprot
      /{}.xml'
```

The output should be something like this:

```
https://www.uniprot.org/uniprot/
    A2AGL3.xml
https://www.uniprot.org/uniprot/
    B0LPN4.xml
https://www.uniprot.org/uniprot/
    E9PZQ0.xml
```

...

We can try to use these links in our internet browser to check if those displayed URLs are working correctly.

Now that we have the URLs, we can automatically download the files using the `curl` command instead of `echo`:

```
$ cat chebi_27732_xrefs_UniProt
    _relevant_identifiers.csv
    | xargs -I {} curl 'https
      ://www.uniprot.org/uniprot
      /{}.xml' -o 'chebi_27732_
      {}.xml'
```

We should note that we now use the `-o` option to save the output to a given file, named after each protein identifier. The equivalent long form to the `-o` option is `--output <file>`.

To check if everything worked as expected we can use the `ls` command to view which files were created:

```
$ ls chebi_27732_*.xml
```

The asterisk character (*) character is here used to represent any file whose name starts with `chebi_27732_` and ends with `.xml`.

To check the contents of any of them, we can use the `less` command:

```
$ less chebi_27732_P21817.xml
```

File Header

We should note that the content of every file has to start with `<?xml` otherwise there was a download error, and we have to run `curl` again for those entries. To

check the header of each file, we can use the `head` command together with `less`.

```
$ head -n 1 chebi_27732_*.xml |  
less
```

The `-n` option specifies how many lines to print, in the previous command just one.

If for any reason, we are not able to download the files from UniProt, we can get them from the book file archive⁴⁵.

Variable

We can now update our script file named *getproteins.sh* to contain the following lines:

```
1 ID=$1 # The CHEBI identifier  
    given as input is renamed  
    to ID  
2 rm -f chebi\_ $ID\_*.xml #  
    Removes any previous files  
3 curl -s "https://www.ebi.ac.uk  
    /chebi/viewDbAutoXrefs.do?  
    d-1169080-e=1&6578706f7274  
    =1&chebiId=$ID&dbName=  
    UniProt" | \  
4 grep -e 'CC - MISCELLANEOUS' -  
    e 'CC - DISRUPTION  
    PHENOTYPE' -e 'CC -  
    DISEASE' | \  
5 gawk -F, '{ print $1 }' |  
    xargs -I {} curl 'https://  
    www.uniprot.org/uniprot  
    /{}.xml' -o chebi\_ $ID\_   
    {}.xml
```

We should note that the last line now includes the `xargs` and `curl` commands, and the `\$ID` variable. This new variable is created in the first line to contain the first value given as argument (`\$1`). So, every time we mention `\$ID` in the script we are mentioning the first value given as argument. This avoids ambiguity in cases where `\$1` is used for other purposes, like in the

gawk command. Since the preceding character of `\$ID` is an underscore (`_`), we have to add a backslash (`\\`) before it. The second line uses the `rm` command to remove any files that were downloaded in a previous execution. We also now added two comments after the hash character, so we humans do not forget why these commands are needed for.

To execute the script once more:

```
$ ./getproteins.sh 27732
```

And again, to check the results:

```
$ head -n 1 chebi_27732_*.xml |  
less
```

XML Processing

Assuming that our information need only concerns human diseases, we have to process the XML file of each protein to check if it represents a *Homo sapiens* (*Human*) protein.

Human Proteins

For performing this filter, we can again use the `grep` command, to select only the lines of any XML file that specify the organism as *Homo sapiens*:

```
$ grep '<name type="scientific">  
Homo sapiens</name>'  
chebi_27732_*.xml
```

We should get in our display the filenames that represent a human protein, i.e. something like this:

```
chebi_27732_P21817.xml:<name  
type="scientific">Homo  
sapiens</name>  
chebi_27732_Q15413.xml:<name  
type="scientific">Homo  
sapiens</name>  
chebi_27732_Q8N490.xml:<name  
type="scientific">Homo  
sapiens</name>  
chebi_27732_Q92736.xml:<name  
type="scientific">Homo  
sapiens</name>
```

We should note that since the asterisk character (*) provides multiple files as

argument to `grep`, the ones whose name starts with `chebi_27732_` and ends with `.xml`, the output now includes the filename (followed by a colon) where each line was matched.

We can use the `gawk` command to extract only the filename, but `grep` has the `-l` option to just print the filename:

```
$ grep -l '<name type="
    scientific">Homo sapiens</
    name>' chebi_27732_*.xml
```

The equivalent long form to the `-l` option is `--files-with-matches`.

The output will now show only the filenames:

```
chebi_27732_P21817.xml
chebi_27732_Q15413.xml
chebi_27732_Q8N490.xml
chebi_27732_Q92736.xml
```

These four files represent the four Human proteins related to *caffeine*.

PubMed Identifiers

Now we need to extract the PubMed identifiers from these files to retrieve the related publications. For example, if we execute the following command:

```
$ grep '<dbReference type="
    PubMed" '
    chebi_27732_P21817.xml
<dbReference type="PubMed" id="
    2298749"/>
<dbReference type="PubMed" id="
    1354642"/>
<dbReference type="PubMed" id="
    8220422"/>
<dbReference type="PubMed" id="
    8661021"/>
<dbReference type="PubMed" id="
    15057824"/>
```

...
To extract just the identifier, we can again use the `gawk` command:

```
$ grep '<dbReference type="
PubMed"'
chebi_27732_P21817.xml |
gawk -F\" '{ print $4 }'
```

We should note that " is used as the separation character and, since the PubMed identifier appears after the third ", the \ \$4 represents the identifier.

Now the output should be something like this:

```
2298749
1354642
8220422
8661021
15057824
...
```

PubMed Identifiers Extraction

Now to apply to every protein we may again use the xargs command:

```
$ grep -l '<name type="
scientific">Homo sapiens</
name>' chebi_27732_*.xml |
xargs -I {} grep '<
dbReference type="PubMed"'
{} | gawk -F\" '{ print
$4 }'
```

This may provide a long list of PubMed identifiers, including repetitions since the same publication can be cited in different entries.

Duplicate Removal

To help us identify the repetitions, we can add the sort command (man sort or sort --help for more information), which will display the repeated identifiers in consecutive lines (due by sorting all identifiers):

```
$ grep -l '<name type="
scientific">Homo sapiens</
name>' chebi_27732_*.xml |
xargs -I {} grep '<
dbReference type="PubMed"'
{} | gawk -F\" '{ print
$4 }' | sort | less
```

For example some repeated PubMed identifiers that we should easily be able to see:

```
10051009
10051009
10097181
10097181
10484775
10484775
```

...

Fortunately, we also have the `-u` option that removes all these duplicates:

```
$ grep -l '<name type="
scientific">Homo sapiens</
name>' chebi_27732_*.xml |
xargs -I {} grep '<
dbReference type="PubMed"'
{} | gawk -F\" '{ print
$4 }' | sort -u
```

To easily check how many duplicates were removed, we can use the word count `wc` command with and without the usage of the `-u` option:

```
$ grep -l '<name type="
scientific">Homo sapiens</
name>' chebi_27732_*.xml |
xargs -I {} grep '<
dbReference type="PubMed"'
{} | gawk -F\" '{ print
$4 }' | sort | wc

$ grep -l '<name type="
scientific">Homo sapiens</
name>' chebi_27732_*.xml |
xargs -I {} grep '<
dbReference type="PubMed"'
{} | gawk -F\" '{ print
$4 }' | sort -u | wc
```

In case we have in our folder any auxiliary file, such as `chebi_27732_P21817_entry.xml`, we should add the option `--exclude *entry.xml` to the first `grep` command.

The output should be something like:

```
255 255 2243
```

```
129 129 1136
```

`wc` prints the numbers of lines, words, and bytes, thus in our case we are interested in first number (`man wc` or `wc --help` for more information). We can see that we have removed $255 - 129 = 126$ duplicates.

Just for curiosity, we can also use the shell to perform simple mathematical calculations using the `expr` command:

```
$ expr 255 - 129
```

Now let us create a script file named `getpublications.sh` by using a text editor to add the following lines:

```
1 ID=$1 # The CHEBI identifier
    given as input is renamed
    to ID
2 grep -l '<name type="
    scientific">Homo sapiens</
    name>' chebi\_$_ID\_*.xml |
    \
3 xargs -I {} grep '<dbReference
    type="PubMed"' {} | \
4 gawk -F\" '{ print $4 }' |
    sort -u
```

Again, do not forget to save it in our working directory, and add the right permissions with `chmod` as we did previously with the other scripts.

To execute the script again:

```
$ ./getpublications.sh 27732
```

We can verify how many unique publications were obtained by using the `-l` option of `wc`, that provides only the number of lines:

```
$ ./getpublications.sh 27732 |
    wc -l
```

The output will be 129 as expected.

Complex Elements

Not always the XML elements are in the same line, as fortunately was the case of the PubMed identifiers. In those cases, we may have to use the `xmllint` command, a parser that is able to extract data through the specification of a XPath query, instead of using a single line pattern as in `grep`.

XPath

XPath (XML Path Language) is a powerful tool to extract information from XML and HTML documents by following their hierarchical structure. Check W3C for more about XPath syntax⁴⁶. We should note that `xmllint` may not be installed by default depending on our operating system, but it should be very easy to do it⁴⁷. If we are using MobaXterm, then we need to install the `xmllint` plugin⁴⁸.

Namespace Problems

In the case of our protein XML files, we can see that their second line defines a specific namespace using the `xmlns` attribute⁴⁹:

```
<uniprot xmlns="http://uniprot.org/uniprot" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://uniprot.org/uniprot http://www.uniprot.org/support/docs/uniprot.xsd">
```

This complicates our XPath queries, since we need to explicitly specify that we are using the local name for every element in a XPath query. For example, to get the data in each reference element:

```
$ xmllint --xpath "//*[local-name()='reference']" chebi_27732_P21817.xml
```

We should note that `//` means any path in the XML file until reaching a reference element. The square brackets in XPath queries normally represent conditions that need to be verified.

Only Local Names

If we are only interested in using local names there is a way to avoid the usage of `local-name()` for every element in a XPath query. We can identify the top-level element, in our case `entry`, and extract all the data that it encloses using a XPath query. For example, we can create the auxiliary file `chebi_27732_P21817_entry.xml` by adding the redirection operator:

```
$ xmllint --xpath "//*[local-
    name()='entry']"
    chebi_27732_P21817.xml >
    chebi_27732_P21817_entry.
    xml
```

The new XML file now starts and ends with the `entry` element without any namespace definition:

```
<entry dataset="Swiss-Prot"
    created="1991-05-01" modified
    ="2018-09-12" version="211">
<accession>P21817</accession>
...
</sequence>
</entry>
```

Now we can apply any XPath query, for example `//reference`, on the auxiliary file without the need to explicitly say that it represents a local name:

```
$ xmllint --xpath '//reference
    '
    chebi_27732_P21817_entry.
    xml
```

The output should contain only the data inside of each reference element:

```

<reference key="1">
  <citation type="journal article"
    date="1990" name="J. Biol.
    Chem." volume="265" first="
    2244" last="2256">
    <title>Molecular cloning of cDNA
      encoding human and rabbit
      forms of the Ca2+ release
      channel (ryanodine receptor)
      of skeletal muscle
      sarcoplasmic reticulum.</
      title>
    ...
    <dbReference type="DOI" id="
      10.1111/cge.12810"/>
  </citation>
  <scope>VARIANTS CCD PRO-2963 AND
    ASP-4806</scope>
</reference>

```

Queries

The XPath syntax allow us to create many useful queries, such as:

- //dbReference – elements of type dbReference that are descendants of something; Result:


```

        <dbReference type="NCBI
          Taxonomy" id="9606"/>
        ...
        <dbReference type="PubMed" id="
          27586648"/>
      
```
- /entry//dbReference – equivalent to the previous query but specifying that the dbReference elements are descendants of the entry element;
- /entry/reference/citation/dbReference – equivalent to the previous query but specifying the full path in the XML file;
- //dbReference/* – any child elements of a dbReference element; Result:

```
<property type="protein
sequence ID" value="
AAA60294.1"/> ... <property
type="match status" value=
"5"/>
```

- //dbReference/property[1] – first property element of each dbReference element; Result:

```
<property type="protein
sequence ID" value="
AAA60294.1"/> ... <property
type="entry name" value="
MIR"/>
```

- //dbReference/property[2] – second property element of each dbReference element; Result:

```
<property type="molecule type"
value="mRNA"/> ...
<property type="match
status" value="5"/>
```

- //dbReference/property[3] – third property element of each dbReference element; Result:

```
<property type="molecule type"
value="Genomic_DNA"/> ...
<property type="project"
value="UniProtKB"/>
```

- //dbReference/property/@type – all type attributes of the property elements; Result:

```
type="protein sequence ID" type
="molecule type" type="
protein sequence ID" ...
type="entry name" type="
match status"
```

- //dbReference/property[@type="protein sequence ID"] – the previous property elements that have an attribute type equal to *protein sequence ID*; Result:

- ```
<property type="protein
 sequence ID" value="AAA60294
 .1"/> ... <property type="
 protein sequence ID" value="
 ENSP00000352608"/>
```
- `//dbReference/property[@type="protein sequence ID"]/@value` – the string assigned to each attribute *value* of the previous property elements; Result:  

```
value="AAA60294.1" value="
AAC51191.1" ... value="
ENSP00000352608"
```
  - `//sequence/text()` – the contents inside the sequence elements; Result:  

```
MGDAEGEDEEVQFLRTDDEVVLQCSATVLKEQLKLC

LAAEGFGNRLCFLEPTSNAQNVPPD
...
LEEHNLANYMFFLMYLINKDETEHTGQESYVWKMY

QERCWDFFPAGDCFRKQYEDQLS
```

We should note that to try the previous queries we only need to replace the string after the `--xpath` option of the previous `xmllint` command, such as:

```
$ xmllint --xpath '//dbReference
 ' chebi_27732_P21817_entry
 .xml
```

Thus, an alternative way to extract the PubMed identifiers using `xmllint` instead of `grep`, would be something like this:

```
$ xmllint --xpath '//dbReference
 [@type="PubMed"]/@id'
$ chebi_27732_P21817_entry.xml
```

However, the output contains all identifiers in the same line and with the id label:

```
id="2298749" id="1354642" id="
8220422" ...
```

## Extracting XPath Results

To extract the identifiers, we need to apply the `tr` command to split the output in multiple lines (one line per identifier), and then the `gawk` command:

```
$ xmllint --xpath '//dbReference
 [@type="PubMed"]/@id'
chebi_27732_P21817_entry.
xml | tr ' ' '\n' | gawk -
F\" '{ NF >0 ; print $2 }'
```

The `tr` command replaces each space by a newline character, and the `gawk` command extracts the value inside the double quotes. We should note that `NF >0` is used to only select lines with at least a separation character `"`, i.e. in our case it ignores empty lines.

---

## Text Retrieval

Now that we have all the PubMed identifiers, we need to download the text included in the titles and abstracts of each publication.

### Publication URL

To retrieve from the UniProt citations service the publication entry of a given identifier, we can again use the `curl` command and a link to the publication entry. For example, if we click on the Format button of the UniProt citations service entry<sup>50</sup>, we can get the link to the RDF/XML version. RDF<sup>51</sup> is a standard data model that can be serialized in a XML format. Thus, in our case, we can deal with this format like we did with XML.

We can retrieve the publication entry by executing the following command:

```
$ curl https://www.uniprot.org/
 citations/1354642.rdf
```

Thus, we can now update the script *getpublications.sh* to have the following commands:

```

1 ID=$1 # The CHEBI identifier
 given as input is renamed
 to ID
2 rm -f chebi_$_ID_*.rdf #
 Removes any previous files
3 grep -l '<name type="
 scientific">Homo sapiens</
 name>' chebi_$_ID_*.xml |
 \
4 xargs -I {} grep '<dbReference
 type="PubMed"' {} | \
5 gawk -F\" '{ print $4 }' |
 sort -u | \
6 xargs -I {} curl 'https://www.
7 uniprot.org/citations/{}.
 rdf'
8 -o chebi_$_ID_{}.rdf

```

We should note that only the second and last lines were updated to remove and retrieve the files, respectively.

Now let us execute the script:

```
$./getpublications.sh 27732
```

It may take a while to download all the entries, but probably no more than one minute with a standard internet connection.

To check if everything worked as expected we can use the `ls` command to view which files were created:

```
$ ls chebi_27732_*.rdf
```

If for any reason, we are not able to download the abstracts from UniProt, we can get them from the book file archive<sup>52</sup>.

## Title and Abstract

Each file has the title and abstract of the publication as values of the `title` and `rdfs:comment` elements, respectively. To extract them we can again use the `grep` command:

```
$ grep -e '<title>' -e '<rdfs:
 comment>'
 chebi_27732_1354642.rdf
```

The output should be something like these two lines:

```
<title>Polymorphisms ...
 hyperthermia.</title>
<rdfs:comment>Twenty-one ...
 gene.</rdfs:comment>
```

To remove the XML elements, we can again use gawk:

```
$ grep -e '<title>' -e '<rdfs:
 comment>'
 chebi_27732_1354642.rdf |
 gawk -F'[<>]' '{ print $3
 }'
```

We should note that we now use two characters as field separators < and > to get the text between the first > and the second <. The first field separator is < so \ \$2 contains the string title or rdfs:comment while \ \$1 is empty. The second field separator is > so \ \$3 contains the string we want to keep.

The output should now be free of XML elements:

```
Polymorphisms ... hyperthermia.
Twenty-one ... gene.
```

Thus, let us create the script *gettext.sh* to have the following commands:

```
1 ID=$1 # The CHEBI identifier
 given as input is renamed
 to ID
1 grep -e '<title>' -e '<rdfs:
 comment>' chebi_ $ID_*.
 rdf | \
2 gawk -F'[<>]' '{ print $3 }'
```

Again do not forget to save it in our working directory, and add the right permissions.

Now to execute the script and see the retrieved text:

```
$./gettext.sh 27732 | less
```

We can save the resulting text in a file named *chebi\_27732.txt* that we may share or read using our favorite text editor, by adding the redirection operator:

```
$./gettext.sh 27732 >
 chebi_27732.txt
```

## Disease Recognition

Instead of reading all that text to find any disease related with *caffeine*, we can try to find sentences about a given disease by using grep:

```
$ grep 'malignant hyperthermia'
 chebi_27732.txt
```

To save the filtered text in a file named *chebi\_27732\_hyperthermia.txt*, we only need to add the redirection operator:

```
$ grep 'malignant hyperthermia'
 chebi_27732.txt >
 chebi_27732_hyperthermia.
 txt
```

This is a very simple way of recognizing a disease in text. The next chapters will describe how to perform more complex text processing tasks.

---

## Further Reading

If we really want to become an expert in shell scripting we may be interested in reading a book specialized in the subject, such as the book entitled *The Linux command line: a complete introduction* (Shotts Jr [2012](#)).

A more pragmatic approach is to explore the vast number of online tutorials about shell scripting and web technologies, such as the ones provided by W3Schools<sup>[53](#)</sup>.



**Open Access** This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.

---

## References

Shotts WE Jr (2012) *The Linux command line: a complete introduction*. No Starch Press, San Francisco

---

## Footnotes

1 <https://en.wikipedia.org/wiki/Caffeine>

2 <https://www.ebi.ac.uk/chebi/searchId.do?chebiId=CHEBI:27732>

3 <http://www.ebi.ac.uk/chebi/>

- 4 <http://www.ebi.ac.uk/chebi/displayAutoXrefs.do?chebiId=CHEBI:27732>
- 5 <http://www.ebi.ac.uk/chebi/viewDbAutoXrefs.do?dbName=UniProt&chebiId=27732>
- 6 [https://web.expasy.org/docs/userman.html#CC\\_line](https://web.expasy.org/docs/userman.html#CC_line)
- 7 [https://en.wikipedia.org/wiki/Text\\_editor](https://en.wikipedia.org/wiki/Text_editor)
- 8 <http://labs.rd.ciencias.ulisboa.pt/book/>
- 9 <http://www.uniprot.org/uniprot/P21817>
- 10 <http://www.uniprot.org/uniprot/E9PZQ0>
- 11 <https://www.uniprot.org/uniprot/P21817/publications>
- 12 <https://www.uniprot.org/citations/1354642>
- 13 <http://labs.rd.ciencias.ulisboa.pt/mer/>
- 14 [http://purl.obolibrary.org/obo/DOID\\_8545](http://purl.obolibrary.org/obo/DOID_8545)
- 15 <https://en.wikipedia.org/wiki/Hyperthermia#Treatment>
- 16 <https://www.ebi.ac.uk/chebi/chebiOntology.do?chebiId=35337>

- 17 [https://en.wikipedia.org/wiki/Bash\\_\(Unix\\_shell\)](https://en.wikipedia.org/wiki/Bash_(Unix_shell))
- 18 <https://docs.microsoft.com/en-us/windows/wsl/about>
- 19 <https://mobaxterm.mobatek.net/>
- 20 <https://en.wikipedia.org/wiki/Spreadsheet>
- 21 [https://en.wikipedia.org/wiki/Path\\_\(computing\)](https://en.wikipedia.org/wiki/Path_(computing))
- 22 <https://www.howtogeek.com/261383/how-to-access-your-ubuntu-bash-files-in-windows-and-your-windows-system-drive-in-bash/>
- 23 <https://mobaxterm.mobatek.net/documentation.html>
- 24 <https://en.wikipedia.org/wiki/Control>
- 25 [https://en.wikipedia.org/wiki/End-of-Transmission\\_character](https://en.wikipedia.org/wiki/End-of-Transmission_character)
- 26 <https://en.wikipedia.org/wiki/Newline>
- 27 <https://notepad-plus-plus.org/>
- 28 [https://www.gnu.org/software/bash/manual/html\\_node/Redirections.html](https://www.gnu.org/software/bash/manual/html_node/Redirections.html)
- 29 [https://en.wikipedia.org/wiki/APT\\_\(Debian\)](https://en.wikipedia.org/wiki/APT_(Debian))
- 30 [https://en.wikipedia.org/wiki/Yum\\_\(software\)](https://en.wikipedia.org/wiki/Yum_(software))

31 apt install curl

32 apt install ca-certificates

33 [https://www.gnu.org/software/bash/manual/html\\_node/Redirections.html](https://www.gnu.org/software/bash/manual/html_node/Redirections.html)

34 <https://www.ebi.ac.uk/chebi/searchId.do?chebiId=CHEBI:15377>

35 <https://www.ebi.ac.uk/chebi/searchId.do?chebiId=CHEBI:30050>

36 <http://www.gnu.org/software/gawk/>

37 apt install gawk

38 [https://en.wikipedia.org/wiki/AWK#Versions\\_and\\_implementations](https://en.wikipedia.org/wiki/AWK#Versions_and_implementations)

39 <https://en.wikipedia.org/wiki/AWK>

40 <https://www6.software.ibm.com/developerworks/education/au-gawk/au-gawk-a4.pdf>

41 <https://www.uniprot.org/help/api>

42 [https://www.uniprot.org/help/api\\_retrieve\\_entries](https://www.uniprot.org/help/api_retrieve_entries)

43 apt install findutils



44 In some versions the scripts may have to use `xargs.exe` to invoke the new version. Or rename the `xargs` shortcut in the bin folder to other name, that way the right version will always be invoked.

45 <http://labs.rd.ciencias.ulisboa.pt/book/>

46 [https://www.w3schools.com/xml/xpath\\_syntax.asp](https://www.w3schools.com/xml/xpath_syntax.asp)

47 `apt install libxml2-utils`

48 <https://mobaxterm.mobatek.net/plugins.html>

49 [https://www.w3schools.com/xml/xml\\_namespaces.asp](https://www.w3schools.com/xml/xml_namespaces.asp)

50 <https://www.uniprot.org/citations/1354642>

51 <https://www.w3.org/RDF/>

52 <http://labs.rd.ciencias.ulisboa.pt/book/>

53 <https://www.w3schools.com/>

## 4. Text Processing

Francisco M. Couto<sup>1</sup>

- (1) LASIGE, Department of Informatics, Faculdade de Ciências, Universidade de Lisboa, Lisbon, Portugal

### Abstract

In the previous chapter we were able to automatically process structured data to retrieve biomedical text about any chemical compound, such as *caffeine*. This chapter will provide a step-by-step introduction to how we can process that text using shell script commands, specifically extract information about diseases related to *caffeine*. The goal is to equip the reader with an essential set of skills to extract meaningful information from any text.

**Keywords** NLP: Natural Language Processing – Text mining – Pattern matching – String matching – Word matching – Evaluation metrics – Regular expressions – Tokenization – NER: Named-Entity Recognition – Relation extraction

---

In the previous chapter we were able to automatically process structured data to retrieve biomedical text about any chemical compound, such as *caffeine*. This chapter will provide a step-by-step introduction to how we can process that text using shell script commands, specifically extract information about diseases related to *caffeine*. The goal is to equip the reader with an essential set of skills to extract meaningful information from any text.

---

### Pattern Matching

We used the `grep` command in the last chapter to find a disease in the text, since `grep` receives as argument a pattern to find an exact match in the text, like

any search functionality provided by conventional text editors. However, we may need to search for multiple patterns even when interested in a single disease. For example, when searching for mentions of *malignant hyperthermia*, we may also be interested in finding mentions using related expressions, such as:

*MH* – acronym

*MHS* – acronym for *malignant hyperthermia susceptible*

Since we already know how to deal with multiple patterns by using the `-e` option, we may easily solve this problem by executing:

```
$ grep -e 'malignant
hyperthermia' -e 'MH' -e '
MHS' chebi_27732.txt
```

## Case Insensitive Matching

When dealing with text, using a case sensitive search is usually a good approach to avoid wrong matches. For example, acronyms are normally in upper case, while the full name is usually in lowercase having sometimes the first letter of each word (or only the first word) in uppercase. So, instead of using a full case sensitive `grep`, we might think on performing a case sensitive `grep` for the acronyms and a case insensitive `grep` for the disease words using the `-i` option:

```
$ grep -e 'MH' -e 'MHS'
chebi_27732.txt
$ grep -i -e 'malignant
hyperthermia' chebi_27732.
txt
```

The equivalent long form to the `-i` option is `--ignore-case`. We should note that each execution of `grep` will produce two separate lists of matching lines that might be overlapped.

Alternatively, we can also convert it to just one case sensitive `grep`, if we are sure that *Malignant hyperthermia* is the only alternative case to *malignant hyperthermia* present in the text. So, we can add it as another pattern:

```
$ grep -e 'Malignant
hyperthermia' -e '
malignant hyperthermia'
-e 'MH' -e 'MHS' chebi_27732.
txt
```

## Number of Matches

To be sure that we are not losing any match, we can count the number of matching lines for both cases. First we execute a case insensitive `grep` and then we execute a case sensitive `grep`, both using the `-c` option:

```
$ grep -c -i 'malignant
hyperthermia' chebi_27732.
txt
$ grep -c -e 'malignant
hyperthermia' -e '
Malignant hyperthermia'
chebi_27732.txt
```

The equivalent long form to the `-c` option is `--count`.

In our case, the output should show 96 and 95 matching lines for the insensitive and sensitive patterns, respectively.

This means that there is a line that is not caught by the case sensitive pattern. To identify which one is, we can manually analyze each of the 96 matching lines one by one. But the goal of this book is exactly avoiding these type of tedious tasks. One thing we can do to solve this issue is to find from the case insensitive matches the one that do not match the case sensitive patterns.

## Invert Match

Fortunately, the `grep` command has the `-v` option that inverts the matching and returns the lines of text that do not contain any matching. The equivalent long form to the `-v` option is `--invert-match`.

Thus, if we apply the inverted match with the case sensitive patterns to the output given by the case insensitive matching, we will get our outlier mention:

```
$ grep -i 'malignant
hyperthermia' chebi_27732.
txt | grep -v -e '
Malignant hyperthermia' -e
'malignant hyperthermia'
```

From the output, we can easily identify the missing matching line:

```
...gene are associated with
Malignant Hyperthermia (MH)
and...
```

We were missing the case where both words have the first letter in uppercase.

Thus, to obtain all the matching lines in a case sensitive match we just have to include the missing match as another pattern:

```
$ grep -c -e 'malignant
hyperthermia' -e '
Malignant hyperthermia' -e
'Malignant Hyperthermia'
chebi_27732.txt
```

## File Differences

Another alternative to compare different matches, is to use the `diff` command that receives as input two files and identifies their differences. So, we can create two auxiliary files and then apply the `diff` to them:

```
$ grep -i 'malignant
hyperthermia'
chebi_27732.txt >
insensitive.txt
$ grep -e 'Malignant
hyperthermia'
-e 'malignant hyperthermia'
chebi_27732.txt > sensitive
.txt
$ diff sensitive.txt insensitive
.txt
```

The output should be the same text.

A problem that may occur with case sensitive matching is that some acronyms are defined with lowercase letters in the middle, such as ChEBI, and humans are not consistent with the way they mention them. The same acronym may be mentioned in their original form or with all letters in uppercase, or just some of them. Moreover, these inconsistent mentions sometimes may even be found in the same publication. We hope not in this book ! ☺

## Evaluation Metrics

These inconsistencies made by humans when mentioning case sensitive expressions, is one of the reasons that most online search engines use case insensitive searches as default. This type of approach favors recall, while case sensitive search favor precision<sup>1</sup>.

Recall is the proportion of the number of correct matches found by our tool

over the total number of correct mentions in the texts (found or not found). Case insensitive searches avoid missing mentions, so they favor recall.

Precision is the proportion of the number of correct matches found by our tool over the total number of matches found (correct or incorrect). Case sensitive searches avoid incorrect matches, so they favor precision.

Normally, there is a trade-off between precision and recall. Using a technique that improves precision, most of the times, will decrease recall, and vice-versa. To know how good the trade-off is, we can use the F-measure, which is the harmonic average of the precision and recall<sup>2</sup>.

## Word Matching

Acronyms (or terms) may also appear inside common words or longer acronyms. For example, when searching for *MH*, the word *victimhood* will produce a match:

```
$ echo "victimhood" | grep -i 'MH'
```

The problem with *victimhood* could be easily solved by using case sensitive matching, but not for a longer acronym. For example, the acronym *NEDMHM* for *neurodevelopmental disorder with midbrain and hindbrain malformations* will produce a case sensitive match:

```
$ echo "NEDMHM" | grep 'MH'
```

One way to address this problem is to use the `-w` option of `grep` to only match entire words, i.e. the match must be preceded and followed by characters that are not letters, digits, or an underscore (or be at the beginning or end of the line). The equivalent long form to the `-w` option is `--word-regexp`.

Using this option, neither *victimhood* or *NEDMHM* will produce a match:

```
$ echo "victimhood" | grep -w -i 'MH'
```

```
$ echo "NEDMHM" | grep -w -i 'MH'
```

Word matching improves precision but decreases recall, since we may miss some less common acronyms that we are not aware of, but are still relevant for our study. For example, consider that we may also be interested in the following acronyms:

*MHE* – acronym for *malignant hyperthermia equivocal*

*MHN* – acronym for *malignant hyperthermia normal*

If we apply word matching, we will not get a match, since both exact

matches are followed by a letter:

```
$ echo "MHE and MHN" | grep -w -
i 'MH'
```

These are not trivial problems to solve by exact pattern matching, we may need regular expressions to address some of these issues more efficiently.

---

## Regular Expressions

When dealing with natural language text we may need more flexibility than the one provided by exact matching. Regular expressions are an efficient tool to extend exact matching with flexible patterns, that may find different matches. As an example, we may be interested in finding all the mentions of the acronym MHS or MHN in a text. For doing that, regular expressions provide the alternation operator that helps us to solve this issue easily by specifying multiple alternatives to match in a specific part of the pattern, in this case an *S* or an *N* as the last character.

Regular expressions can be better understood by clearly separating three distinct components:

*input* – any string where we want to find something

*pattern* – a string that specifies what we are looking for

*match* – a fragment of the input (a substring) where the pattern can be found

In our examples, the input is the text file *chebi\_27732.txt*, but it can be the amino acid sequences that we previously extracted from the UniProt file entries. Until now the pattern has represented an exact string to look for, where each match is an exact replica of the pattern occurring at a given position of the input string. When using regular expressions, the pattern contains special characters, whose purpose are not to directly match with the input but instead have a special meaning. These special characters represent operators that specify which different types of strings we want to find in the input. For example, strings that start with *MH* and end with *S* or an *N*. By using regular expressions, the matches are not replicas of the pattern, they can be different strings as long as they satisfy the specified pattern.

## Extended Syntax

The `grep` command allows us the possibility to include regular expression operators in the input pattern. `grep` understands two different versions of



regular expression syntax: basic and extended<sup>3</sup>. We will use the extended syntax for two reasons: (i) the basic does not support relevant operators, such as alternation; (ii) and to clearly differentiate exact matching from regular expression matching. Thus, instead of the `-e` option previously used in the `grep` command, we will start to use the `-E` option, which makes the command interpret the pattern as an extended regular expression. The equivalent long form to the `-E` option is `--extended-regexp`. We should note that this option does not affect the matching when using a pattern without any regular expression operator, such as `MH`. For example, the following commands will produce the same results:

```
$ echo -e 'MHS\nMHN' | grep -e
 'MH'
$ echo -e 'MHS\nMHN' | grep -E
 'MH'
```

Note, that we use the `-e` option so the `echo` command interprets the `\\n` characters as a newline. Thus, the `echo` command outputs two lines, that are given as input to the `grep` command. We should note that the `grep` command filters lines.

## Alternation

The first regular expression operator we will test is the alternation, which we introduced above. An alternation is represented by the bar character (`|`) that specifies a pattern where any match must include either the preceding or following characters. The preceding and following characters can be enclosed within parentheses to better specify the scope of the alternation operator. For example, the pattern for finding strings that start with `MH` and end with `S` or an `N` can be written as:

```
$ echo -e 'MHS\nMHN' | grep -E
 'MH(S|N)'
```

## Basic Syntax

If we use the basic regular expression syntax no match will be found, since the alternation operator is not supported:

```
$ echo -e 'MHS\nMHN' | grep -e
 'MH(S|N)'
```

We will have a match only if the `|` and the parentheses are in the input string, since it is not interpreted as an operator:



```
$ echo -e 'MH(S|N)' | grep -e
 'MH(S|N)'
```

## Scope

To better understand the scope of an alternation, we can remove the parentheses from the pattern and add the `-w` option:

```
$ echo -e 'MHS\nMHN' | grep -w
 -E 'MHS|N'
```

We only get the first line. This is explained because the alternation operator is applied to all the preceding characters, i.e. the `grep` will search for the *MHS* word or the *N* word. If we add a single *N* to the input string we already get another match:

```
$ echo -e 'MHS\nN' | grep -w -E
 'MHS|N'
```

We can also move the opening parenthesis one character to the left:

```
$ echo -e 'MHS\nMHN' | grep -E
 'M(HS|N)'
```

Only *MHS* is now displayed, since the alternative now represents *MN* without the *H*.

## Multiple Alternatives

We are not limited to two alternatives, we can have multiple `|` operators in a pattern. For example, the following command will find any of the three acronyms *MHS*, *MHE* or *MHN*:

```
$ echo -e 'MHS\nMHN\nMHE' | grep
 -E 'MH(S|N|E)'
```

We can now transform our previous `grep` command with multiple case sensitive patterns:

```
$ grep -c -e 'Malignant
 hyperthermia' -e '
 Malignant Hyperthermia' -e '
 'malignant hyperthermia'
 chebi_27732.txt
```

in a `grep` command with a single pattern using alternation:

```
$ grep -c -E '(M|m)alignant(H|h)
 yperthermia' chebi_27732.
 txt
```

And we will obtain the same 96 matches.

## Multiple Characters

A useful regular expression feature is that we can use the dot character (.) to represent any character, so if we want to find all the acronyms that start with *MH* we can execute the following command:

```
$ grep -o -w -E 'MH.'
 chebi_27732.txt | sort -u
```

We should note that we use the `-o` option of the command `grep` so it just displays the matches and not all the line that includes the match. The equivalent long form to the `-o` option is `--only-matching`.

The output will be the following three-character lines:

```
MH
MH)
MH,
MH.
MH1
MH2
MHE
MHN
MHS
```

If we really want to match only the dot character, we have to precede it with a backslash character (\\):

```
$ grep -o -w -E 'MH\\.'
 chebi_27732.txt | sort -u
```

Now only the *MH.* will be displayed.

We can check that there are some matches that are not really acronyms, such as *MH)* and *MH,*.

## Spaces

We should note that *MH* appears because the space character can also be matched. For example, the following text includes a word match with *MH\_* since the parenthesis is considered a word delimiter character (not a letter, digit or underscore):

```
... susceptible to MH (MHS) ...
```

On the other hand, the following text does not include a word match with *MH\_*:

```
... markers and MH
 susceptibility ...
```

Thus, what we really want is matches where the third character is a letter or a

numerical digit.

Sometimes, the text includes other characters that also represent horizontal or vertical space in typography, such as the tab character. All these characters are known as whitespaces and can be represented by the expression `\\s` in a pattern<sup>4</sup>. The following command demonstrates that both the space and the tab characters are matched by `\\s`:

```
echo -e 'space: :\ntab:\t:' |
grep -E '\\s'
```

## Groups

Fortunately, the regular expressions include the group operator that let us easily specify a set of characters. A group operator is represented by a set of characters enclosed within square brackets. Any of the enclosed characters can be matched.

For example, the previous command to find any of the three acronyms can be replaced by:

```
$ echo -e 'MHS\nMHN\nMHE' | grep
-E 'MH[SNE]'
```

We should note that only one of the three letters, *S*, *N* or *E* will be matched in the input string.

## Ranges

Still, this is not solving our need to only match letters or digit. However, we can also specify characters ranges with the dash character (-). For example, to find all the acronyms that start with *MH* followed by any alphabet letter:

```
$ grep -o -w -E 'MH[A-Z]'
chebi_27732.txt | sort -u
```

This will result in only three acronyms:

MHE

MHN

MHS

We should note that `A-Z` represents any alphabet letter in uppercase, a lowercase letter will not be matched:

```
$ echo -e 'MHS\nMHs' | grep -E '
MH[A-Z]'
```

If we intend to keep the usage of a case sensitive `grep` and at the same time find lowercase matches, then we need to add the `a-z` range:

```
$ echo -e 'MHS\nMHs' | grep -E '
MH[A-Za-z]'
```

We should note that the dot character inside a range represents itself and not

any character:

```
$ echo -e 'MHS\nMH.' | grep -E 'MH[.]'
```

Additionally, to include the acronyms that end with a numerical digit we need to add the 0-9 range:

```
$ grep -o -w -E 'MH[A-Z0-9]'
chebi_27732.txt | sort -u
```

Finally, we have the correct list of all three character acronyms starting with *MH*:

```
MH1
MH2
MHE
MHN
MHS
```

## Negation

Another frequent case is the need to match any character with a few exceptions. For example, if we need to find all the matches that start with *MH* followed by any character except an alphabet letter. Fortunately, we can use the negation feature within a group operator. The negation feature is represented by the circumflex character ( $\wedge$ ) right next to the left bracket. The negation means that all the characters and ranges enclosed within the brackets are the ones that cannot be matched. Thus, a solution to the above example is to add the A-Z range after the circumflex:

```
$ grep -o -w -E 'MH[^A-Z]'
chebi_27732.txt | sort -u
```

We can see that all of the three acronyms *MHS*, *MHE* or *MHN* will be missing from the output:

```
MH
MH,
MH.
MH)
MH1
MH2
```

If we do not want the *MH\_* acronym, we can add the space character to the negative group:

```
$ grep -o -w -E 'MH[^A-Z]'
chebi_27732.txt | sort -u
```

The output should now contain one less acronym:

MH,  
MH.  
MH)  
MH1  
MH2

## Quantifiers

Above we were interested in finding acronyms composed of exactly three characters. However, we may need to find all acronyms that start with *MH* independently of their length. This functionality is also available in regular expressions using the quantifiers operators.

### Optional

The simplest quantifier is the optional operator that is specified by an item followed by the question mark character (?). The item can be a character, an operator or a sub-pattern enclosed by parentheses. That item becomes optional for matching, i.e. a match can either contain that item or not.

For example, to find all the acronyms starting with *MH* and followed by one alphabetic letter or none:

```
$ grep -o -w -E 'MH[A-Z0-9]?'
chebi_27732.txt | sort -u
```

Given that the third character is optional the output will include the two-character acronym *MH*, but not the *MH\_* match:

MH  
MH1  
MH2  
MHE  
MHN  
MHS

We can add the space character to the group:

```
$ grep -o -w -E 'MH[A-Z0-9]?'
chebi_27732.txt | sort -u
```

Now the output includes the two-character acronym *MH* and the *MH\_* match:

MH  
MH  
MH1  
MH2  
MHE  
MHN  
MHS

### ***Multiple and Optional***

To find all the acronyms independently of their length, we can use the asterisk character (\*). The preceding item becomes optional and can be repeated multiple times. For example, to find all the acronyms starting with *MH* and which may be followed any number of alphabetic letters or numeric digits:

```
$ grep -o -w -E 'MH[A-Z0-9]*'
chebi_27732.txt | sort -u
```

The output now includes the four-character acronym *MHS1*:

MH  
MH1  
MH2  
MHE  
MHN  
MHS  
MHS1

We should note that the `grep` command uses a greedy approach, i.e. it will try to match as many characters as possible. For example, the following command will match *MH1* and not *MH*:

```
$ echo 'MH1' | grep -o -E 'MH
[0-9]*'
```

### ***Multiple and Compulsory***

To make the preceding item compulsory and able to repeat it multiple times, we may replace the asterisk by the plus character (+). For example, the following pattern will find all the acronyms starting with *MH* followed by at least one alphabetic letter or numeric digit:

```
$ grep -o -w -E 'MH[A-Z0-9]+'
```

```
chebi_27732.txt | sort -u
```

We should note that the output does not contain the two character acronym *MH*:

MH1  
MH2  
MHE  
MHN  
MHS  
MHS1

## ***All Options***

The above quantifiers are the most popular, but the functionality of all of them can be reproduced by using curly braces to specify the minimal and maximum number of occurrences. The item is followed by an expression of the type `\{n,m\}` where `n` and `m` are to be replaced by a number specifying the minimum and maximum number of occurrences, respectively. `n` and `m` may also be omitted, which means that no minimum or maximum limit is to be imposed.

Using curly brackets, the question mark character (?) can be replaced by `\{0,1\}`. Thus, the following two patterns are equivalent:

```
$ grep -o -w -E 'MH[A-Z0-9]?'
chebi_27732.txt | sort -u

$ grep -o -w -E 'MH[A-Z0-9]{0,1}' chebi_27732.txt
| sort -u
```

The asterisk character (\*) can be replaced by `\{0,\}`. Thus, the following two patterns are equivalent:

```
$ grep -o -w -E 'MH[A-Z0-9]*'
chebi_27732.txt | sort -u

$ grep -o -w -E 'MH[A-Z0-9]{0,}'
chebi_27732.txt | sort -u
```

The plus character (+) can be replaced by `\{1,\}`. Thus, the following two patterns are equivalent:

```
$ grep -o -w -E 'MH[A-Z0-9]+'chebi_27732.txt | sort -u

$ grep -o -w -E 'MH[A-Z0-9]{1,}'
chebi_27732.txt | sort -u
```

On the other hand using `\{1,1\}` is the same as not having any operator. Thus, the following two patterns are equivalent:

```
$ grep -o -w -E 'MH[A-Z0-9]'
 chebi_27732.txt | sort -u
$ grep -o -w -E 'MH[A-Z0
-9]{1,1}' chebi_27732.txt
| sort -u
```

The previous commands display the all the three-character acronyms:

```
MH1
MH2
MHE
MHN
MHS
```

For example, if we are looking for acronyms with exactly 4 characters then we can apply the following pattern:

```
$ grep -o -w -E 'MH[A-Z0
-9]{2,2}' chebi_27732.txt
| sort -u
```

We should note that we use 2 as both the minimum and maximum since *MH* already count as 2 characters.

The output of the previous command is now the four-character acronym:

```
MHS1
```

---

## Position

Sometimes besides the match, we are also interested in limiting the matches to specific parts of the input string. For example, to identify start and stop codons in a protein sequence, we need to limit the matches to the beginning or the end of the sequence. In text, we may for example be interested in lines starting with a name of a disease. To take in account the position of a match regular expressions patterns can start with the circumflex character ( $\wedge$ ) and/or end with the dollar sign character ( $\$$ ).

If the pattern starts with a circumflex then only matches at the beginning of the line will be considered. On the other hand, if the pattern ends with a dollar then only matches at the end of the line will be considered.

## Beginning

For example, if we are looking for lines starting with *Malignant Hyperthermia* we can use the following pattern:



```
$ grep -E '^ (M|m)alignant (H|h)
yperthermia' chebi_27732.
txt
```

The output will include the list of lines beginning with a mention to *Malignant Hyperthermia*:

```
Malignant hyperthermia (MH) is a
potentially fatal autosomal
...
Malignant hyperthermia (MH) is a
pharmacogenetic disorder ...
```

To check how many of the matching lines were filtered, we can count the number of occurrences when using the circumflex and when not:

```
$ grep -c -E '^ (M|m)alignant (H|h)
yperthermia' chebi_27732.
txt
$ grep -c -E ' (M|m)alignant (H|h)
yperthermia' chebi_27732.
txt
```

The output will show that only 23 of the 96 matches were considered.

## Ending

If we are looking for lines ending with a mention to *Malignant Hyperthermia*, then we can add the dollar character to the end of the pattern:

```
$ grep -E ' (M|m)alignant (H|h)
yperthermia.$' chebi_27732
.txt
```

To allow a punctuation character before the end of the line, we added the dot character before the dollar character in the pattern. The dot character matches any character, including the dot itself.

The output will be the list of lines ending with a mention to *Malignant Hyperthermia*:

```
Novel mutation in the RYR1 gene
(R2454C) in a patient with
malignant hyperthermia.
```

Identification of a novel  
mutation in the ryanodine  
receptor gene (RYR1) in  
patients with malignant  
hyperthermia.

Novel skeletal muscle ryanodine  
receptor mutation in a large  
Brazilian family with  
malignant hyperthermia.

\*\*\*  
We can check how many lines were filtered by using again the -c option:

```
$ grep -c -E '(M|m)alignant(H|h)
yperthermia.$' chebi_27732
.txt
$ grep -c -E '(M|m)alignant(H|h)
yperthermia' chebi_27732.
txt
```

The output will show that only 15 of the 96 matches were at the end of the line.

## Near the End

Sometimes we do not want the mention ending exactly at the last character. We may be more flexible and allow a following expression, or a given number of characters. For example, to allow 10 other characters between the end of the line and the mention of *Malignant Hyperthermia*, we can add a quantifier to the dot operator:

```
$ grep -c -E '(M|m)alignant (H|h)
yperthermia.{0,10}$'
chebi_27732.txt
```

The output will show that we have 20 matches.

If we remove the -c option, we will be able to check that words, such as *families* and *patients*, are now allowed to appear between the mention of *Malignant Hyperthermia* and the end of the line:

```

...
Novel mutations in C-terminal
 channel region of the
 ryanodine receptor in
 malignant hyperthermia
 patients.
...
Novel missense mutations and
 unexpected multiple changes
 of RYR1 gene in 75 malignant
 hyperthermia families.
...

```

## Word in Between

To allow a word in between, independently of its length, we can add to the pattern an optional sequence of non-space characters (the word) preceded by a space:

```

$ grep -c -E '(M|m)alignant(H|h)
 yperthermia([^]*)?.$'
 chebi_27732.txt

```

The output will show that we have 24 matches. We should note that the `[\^ ]` operator avoids having two words.

If we remove the `-c` option, we will be able to check that lengthy words (with more than 10 characters), such as *susceptibility*, are now allowed to appear between the mention of *Malignant Hyperthermia* and the end of the line:

```

...
Ryanodine receptor gene point
 mutation and malignant
 hyperthermia susceptibility.
...

```

## Full Line

If we want lines that start with a mention to *Malignant Hyperthermia* and end with an acronym, *MH* or *MHS*, then we can execute two `grep` commands. The first gets the lines starting with *Malignant Hyperthermia* and the next filters the output of the latter with lines ending with an acronym:

```
$ grep -E '^(M|m)alignant (H|h)
yperthermia' chebi_27732.
txt | grep -w -E 'MHS?.$'
```

Alternatively, we can add both the circumflex and dollar operators to the same pattern. However, we cannot forget to add `.*` to match anything in between them, since we are asking full line matches:

```
$ grep -w -E '^(M|m)alignant (H|h)
yperthermia.*MHS?.$'
chebi_27732.txt
```

We can see that both commands match all the text of the abstract since each abstract is stored in a single line of the file:

```
Malignant hyperthermia (MH) is a
pharmacogenetical
complication ... as for
genetic diagnosis of MH.
Malignant hyperthermia
susceptibility (MHS) is a
subclinical pharmacogenetic
disorder ... been tested
positive for MHS.
```

This demonstrates the problem of tokenization, since usually what we really need is to match a full sentence or a phrase. And in that case each line should represent a sentence or phrase from the abstract.

## Match Position

For more advanced processing, we may be interested in knowing the exact position of the matches in a given line. This can be done by using the `-b` option of `grep`, which provides the number of bytes in the line before the start of the match:

```
$ echo 'MHS MHN MHE' | grep -b -
o -w -E 'MH[SNE]'
```

The equivalent long form to the `-b` option is `--byte-offset`.

The output shows the list of matches preceded by their position in the given line:

```
0:MHS
4:MHN
8:MHE
```

---

## Tokenization

As we have shown in the previous section, sometimes we need to work at the level of a sentence and not use a full document as the input string. Tokenization is a Natural Language Processing (NLP) task that aims at identifying boundaries in the text to fragment it into basic units called tokens. These tokens can be sentences, phrases, multi-word expressions, or words.

### Character Delimiters

In most languages, some specific characters can be considered as accurate boundaries to fragment text into tokens. For example, the space character to identify words; the period (.), the question mark (?) and the exclamation mark (!) to identify the ending of a sentence; and the comma (,), the semicolon (;), the colon (:) or any kind of parenthesis to identify a phrase within a sentence. However, this problem may be more complex in languages without explicitly delimiters, such as Chinese (Wu and Fung [1994](#)).

A common approach to tokenization is to use regular expressions to replace these delimiters by newline characters. This will result in a token per line. For example, we can replace the characters specifying the end of a sentence with a newline by using the `tr` command and then count the number of lines:

```
$ tr '[:!?' ' '\n' < chebi_27732.
txt | wc -l
```

We get 1493 lines from the original 248 lines:

```
$ wc -l chebi_27732.txt
```

Unfortunately, this is not just so simple. We need to analyze the output:

```
$ tr '[:!?' ' '\n' < chebi_27732.
txt | less
```

### Wrong Tokens

We can check that: (i) many lines are empty because an extra newline character will be added to the last sentence, and (ii) the dot character is also used as a decimal mark in a number, then some sentences are split in multiple lines because they have decimal number in them. For example, the original sentence:

```
These 10 mutations account for
21.9% of the North American
MH-susceptible population
```

is split in two lines:

```
These 10 mutations account for
21
9% of the North American MH-
susceptible population
```

## String Replacement

This means that looking at just one character is not enough, we need some context. For performing this, we will use the `sed` command that we may consider as a more powerful version of the `tr` command. The `sed` command is a stream editor that can receive as input a string and perform basic text transformations, such as replace one expression by another, that are available in almost all text editors. For example, we can use a simple `sed` to convert every mention of *caffeine* by its ChEBI identifier:

```
$ sed -E 's/caffeine/CHEBI
:27732/gi' chebi_27732.txt
```

The `-E` option allow us to use extended regular expressions, like we used before in `grep`. The `s` option has the following syntax `'s/FIND/REPLACE/FLAGS'`, where: `FIND` is the pattern to find in the input string; `REPLACE` the expression to replace the matches; `FLAGS` are multiple options, such as `g` to replace all matches in each line and not just the first one, and `i` to be case insensitive.

For example, the original fragment of text:

```
... link between the caffeine
threshold and tension ...
```

will be converted to:

```
... link between the CHEBI:27732
threshold and tension ...
```

## Multi-character Delimiters

To replace the delimiter characters by a newline when followed by at least one space character, we can use the following command:

```
$ sed -E 's/[.!?] +/\n/g'
chebi_27732.txt
```

We should note that by making compulsory a space character, we avoid: (i) empty lines by splitting a sentence that is already at the end of the line (assuming there are no ghost space characters at the end of each line), and (ii) decimal markers because they are followed by numerical digits and not spaces.

We now get 1067 lines from the original 248 lines:



```
$ sed -E 's/[.!?] +/\n/g'
chebi_27732.txt | wc -l
```

## Keep Delimiters

The previous `sed` command is removing the delimiter characters from the text, and this may cause other problems. The best solution is to keep the delimiter characters and just add the newline. The `sed` command allows us to keep each match for a specific part of the pattern (sub-pattern) by enclosing it within parentheses. To include the match of a sub-pattern in the replace expression, we can use the backslash and its numerical order. Thus, we can improve our `sed` command by using this technique so we do not remove any delimiter character:

```
$ sed -E 's/([.!?])(+)/\1\n\2/g'
chebi_27732.txt
```

However, other common issues may still persist. For instance, there are some sentences starting right after the delimiter characters without any space in between:

```
... bulk.Fetal ...
... sequencing.Whole ...
```

These sentences include a delimiter character directly followed by an alphabetic letter:

```
$ sed -E 's/([.!?])(+)/\1\n\2/g'
chebi_27732.txt | grep -
i '[.!?][a-z]'
```

To minimize this issue, we can change the pattern so the compulsory space character become optional, but requiring a following uppercase alphabetic letter:

```
$ sed -E 's/([.!?])(*[A-Z])/\\1\\
n\2/g' chebi_27732.txt |
wc -l
```

We now get 1127 lines, i.e. this pattern is more flexible and was able to split more 60 sentences. This does not mean that is free of errors. It is almost impossible to derive a rule that covers all the possible typos humans can produce.

As an example, Fig. 4.1 show a complex pattern adapted from Wikipedia. The pattern is equivalent to `\. {2,} [A-Z]`, and identifies multiples spaces at the beginning of a sentence. The pattern requires at least two spaces to be matched, but only after a period and before an uppercase letter.

I watch three climb before it's my turn. It's a tough one. The guy before me tries twice. He falls twice. After the last one, he comes down. He's finished for the day. It's my turn. My buddy says "good luck!" to me. I noticed a bit of a problem. There's an outcrop on this one. It's about halfway up the wall. It's not a

**Fig. 4.1** Identifying multiple spaces at the beginning of a sentence using regular expressions (Adapted from: [https://en.wikipedia.org/wiki/Regular\\_expression](https://en.wikipedia.org/wiki/Regular_expression))

## Sentences File

Using our previous pattern, we can update our script named *gettext.sh* to provide the text already split in sentences by adding the `sed` command:

```
1 ID=$1 # The CHEBI identifier
 given as input is renamed
 to ID
2 grep -e '<title>' -e '<rdfs:
 comment>' chebi_ $ID_*.
 rdf | \
3 gawk -F' [<>]' '{ print $3 }' |
 \
4 sed -E 's/([.!?])(*[A-Z])/ \1\
 n\2/g'
```

To save the output as a file named *chebi\_27732\_sentences.txt*, we only need to add the redirection operator:

```
$./gettext.sh 27732 >
 chebi_27732_sentences.txt
```

Each line of the file *chebi\_27732\_sentences.txt* represents a sentence.

---

## Entity Recognition

To select the sentences with one of our acronyms, we can use the `grep` command and our sentences file:

```
$ grep -w -E 'MH[SNE]?'
 chebi_27732_sentences.txt
```



The output will only include matching sentences:

```
...
Interestingly, the data suggest
 a link between the caffeine
 threshold and tension values
 and the MH/CCD phenotype.
```

Alternatively, we can use the `-n` option to get the number of the line and the `-o` option to get the acronym matched:

```
$ grep -n -o -w -E 'MH[SNE]?'
 chebi_27732_sentences.txt
```

The equivalent long form to the `-n` option is `--line-number`. The output should be something like this:

```
...
1106:MH
1106:MH
1108:MH
1110:MH
1111:MH
```

We can now make a script that receives a pattern as argument and the input text as the standard input, to display the line numbers and the matches in a TSV format. Thus, let us create a script file named *getentities.sh* with the following lines:

```
1 PATTERN=$1
2 grep -n -o -w -E $PATTERN | \
3 tr ':' '\t'
```

Again we should not forget to save the file in our working directory, and add the right permissions with `chmod`, as we did with our scripts in the previous chapter.

The first line stores the pattern given as argument in the variable `PATTERN`. The `grep` command finds the matches and the `tr` command replaces each colon by a tab character to produce TSV content.

We can now execute the script giving the pattern as argument and the sentences file as standard input:

```
$./getentities.sh 'MH[SNE]?' <
 chebi_27732_sentences.txt
```

The output should be something like this:

```
...
1106 MH
1106 MH
1108 MH
1110 MH
1111 MH
```

We should note that now we have the values separated by a tab character, i.e. the output is in TSV format.

The output can also be saved as a TSV file that we can open directly in our preferred spreadsheet application. For example, to save it as *chebi\_27732.tsv*, we only need to add the redirection operator:

```
$./getentities.sh 'MH[SNE]?' <
 chebi_27732_sentences.txt
> chebi_27732.tsv
```

## Select the Sentence

If we want to analyze a specific matched sentence, we can use a text editor and go to that line number. A more efficient alternative is to use the `print p` option of `sed` to output a given line number. For example, to check the *MHS* match at line 2:

```
$ sed -n '2p'
 chebi_27732_sentences.txt
```

Now we can easily check the context of the match:

```
... in susceptible people (MHS)
 by volatile ...
```

---

## Pattern File

The script created in the previous section only accepts one pattern, however we may need to recognize different entities, or different mentions of the same entity, such as the official name, possible synonyms, and the acronyms. Fortunately, `grep` allows us to include a list of patterns directly from a file using the `-f` option. The equivalent long form to the `-f` option is `--file=FILE`. For example, we can create a text file named *patterns.txt* with the following three patterns:

```
(M|m)alignant (H|h)yperthermia
MH[SNE]?
(C|c)affeine
```

Then we can execute the previous `grep` but using multiple patterns

specified in the pattern file:

```
$ grep -n -o -w -E -f patterns.
txt chebi_27732_sentences.
txt
```

Analyzing the output, we can check that the same sentences may include different entities:

```
1110:MH
1110:caffeine
1111:caffeine
1111:MH
```

We can now update our script named *getentities.sh* to receive as input not a single pattern but the filename where multiple patterns can be found.

```
1 PATTERNS=$1
2 grep -n -o -w -E -f $PATTERNS
| \
3 tr ':' '\t'
```

We can execute the script giving as argument the file containing the patterns:

```
$./getentities.sh patterns.txt
< chebi_27732_sentences.
txt
```

To save the output as a file named *chebi\_27732.tsv*, we only need to add the redirection operator:

```
$./getentities.sh patterns.txt
< chebi_27732_sentences.
txt > chebi_27732.tsv
```

Using the *patterns.txt* file is very useful if for example we are not focused in a single disease, and we want to find any disease mentioned in the text. In these cases, we have to create a file with the full lexicon of diseases. This topic will be addressed in the following chapter.

---

## Relation Extraction

Finding the relevant entities in text is sometimes not enough. We need to know which sentences may describe possible relationships between those entities, such as a relation between a disease and a compound.

This is a complex text mining challenge, but a simple approach is to construct a pattern that allow any kind of characters between two entities:

```
$ grep -n -w -E 'MH[SNE]?.*(C|c)
 affeine'
 chebi_27732_sentences.txt
```

The following sentence is one of the seven displayed sentences mentioning a possible relation:

```
239: ... MHS families were
 investigated with a caffeine
 ...
```

However, we are missing all the sentences that have *caffeine* first:

```
$ grep -n -w -E '(C|c)affeine.*
 MH[SNE]?'
 chebi_27732_sentences.txt
```

We will be able to see that sometimes *caffeine* comes first:

```
801: ... caffeine-halothane
 contracture test were greater
 in those who had a known MH
 ...
1111: ... caffeine threshold and
 tension values and the MH
 ...
```

## Multiple Filters

The most flexible approach is use two `grep` commands. The first selects the sentences mentioning one of the entities, and the other selects from the previously selected sentences the ones mentioning the other entity. For example, we can first search for the acronyms and then for *caffeine*:

```
$ grep -n -w -E 'MH[SNE]?'
 chebi_27732_sentences.txt
 | grep -w -E '(C|c)affeine'
```

This will show all the nine sentences mentioning *caffeine* and an acronym.

## Relation Type

If we are interested in a specific type of relationship, we may have an additional filter for a specific verb. For example, we can add a filter for sentences with the verb *response* or *diagnosed*:

```
$ grep -n -w -E 'MH[SNE]?'
 chebi_27732_sentences.txt
 | grep -w -E '(C|c)affeine
 ' | grep -w -E 'response|
 diagnosed'
```

We should note that this does not take in account where the verb appears in the sentence. For example, in the following sentence the verb *response* appears first than any of the two entities:

```
50: The relationship between the
 IVCT response and genotype
 was ... the number of MHS
 discordants ... at 2.0\,mM
 caffeine ...
```

If the verb needs to appear between the two entities, we have to construct a pattern that have these words in the middle of them:

```
$ grep -n -w -E 'MH[SNE]?.*(
 response|diagnosed) .* (C|c)
 affeine'
 chebi_27732_sentences.txt
```

We can see now that the previous sentence (line 50) is not presented as a match.

## Remove Relation Types

We may also be interested in ignoring specific type of relations. To do that, we only need to use the `-v` (or `--invert-match`) option. For example, to ignore sentences with the word *response* or *diagnosed*:

```
$ grep -n -w -E 'MH[SNE]?'
 chebi_27732_sentences.txt
 | grep -w -E '(C|c)affeine
 ' | grep -v -w -E '
 response|diagnosed'
```

All the resulting sentences do not mention *response* or *diagnosed*.

---

## Further Reading

If we want to have a deeper knowledge about text processing tasks and challenges, we may be interested in reading some chapters of the book entitled *Speech and language processing* (Jurafsky and Martin [2014](#)). The book is a

highly specialized document explaining in full detail the topics here briefly described.

To have an overview about the state-of-art in text processing tools using biomedical literature, we should consider reading a recent and comprehensive survey (Lamurias and Couto 2019).



**Open Access** This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.

---

## References

Jurafsky D, Martin JH (2014) Speech and language processing, vol 3. Pearson, London

Lamurias A, Couto F (2019) Text mining for bioinformatics using biomedical literature. In: Ranganathan S, Nakai K, Schönbach C, Gribskov M (eds) Encyclopedia of bioinformatics and computational biology, vol 1. Elsevier, Oxford

Wu D, Fung P (1994) Improving chinese tokenization with linguistic filters on statistical lexical acquisition. In: Proceedings of the 4th conference on applied natural language processing

---

## Footnotes

1 [https://en.wikipedia.org/wiki/Precision\\_and\\_recall](https://en.wikipedia.org/wiki/Precision_and_recall)

2 [https://en.wikipedia.org/wiki/F1\\_score](https://en.wikipedia.org/wiki/F1_score)

3 <https://www.regular-expressions.info/posix.html>

4 [https://en.wikipedia.org/wiki/Whitespace\\_character](https://en.wikipedia.org/wiki/Whitespace_character)

## 5. Semantic Processing

Francisco M. Couto<sup>1</sup>

- (1) LASIGE, Department of Informatics, Faculdade de Ciências, Universidade de Lisboa, Lisbon, Portugal

### Abstract

In the previous chapter we were able to automatically process text by recognizing a limited set of entities. This chapter will introduce the world of semantics, and present step-by-step examples to retrieve and enhance text and data processing by using semantics. The goal is to equip the reader with the basic set of skills to explore semantic resources that are nowadays available using simple shell script commands.

**Keywords** Ontologies – OWL: Web Ontology Language – Semantic resources – DO: disease ontology – ChEBI: chemical entities of biological interest – Ancestors – Recursion – Lexicons – Entity linking – Semantic similarity

---

### Classes

In the previous chapters we searched for mentions of *caffeine* and *malignant hyperthermia* in text. However, we may miss related entities that may also be of our interest. These related entities can be found in semantic resources, such as ontologies. The semantics of *caffeine* and *malignant hyperthermia* are represented in *ChEBI* and *DO* ontologies, respectively.

### OWL Files

Thus, we can start by retrieving both ontologies, i.e. their OWL files.



```
$ curl -O 'https://raw.githubusercontent.com/
DiseaseOntology/
HumanDiseaseOntology/
master/src/ontology/
releases/2018-11-02/doid.
owl'

$ curl -O 'ftp://ftp.ebi.ac.uk/
pub/databases/chebi/
archive/rell69/ontology/
chebi_lite.owl'
```

The `-O` option saves the content to a local file named according to the name of the remote file, usually the last part of the URL. The equivalent long form to the `-O` option is `--remote-name`.

The previous commands will create the files *chebi\_lite.owl* and *doid.owl*, respectively. We should note that these links are for the specific releases used in this book. Using another release may change the output of the examples presented in this chapter.

The links may also change in the future, so we may need to check them on the BioPortal<sup>1</sup> or on the OBO Foundry<sup>2</sup> webpages. Alternatively, we can also get the OWL files from the book file archive<sup>3</sup>.

## Class Label

Both OWL files use the XML format syntax. Thus, to check if our entities are represented in the ontology, we can search for ontology elements that contain them using a simple `grep` command:

```
$ grep '>malignant hyperthermia
<' doid.owl

$ grep '>caffeine<' chebi_lite.
owl
```

For each `grep` the output will be the line that describes the property label (*rdfs:label*), which is inside the definition of the class that represents the entity:



```
<rdfs:label rdf:datatype="http://www.w3.org/2001/XMLSchema#string">malignant hyperthermia</rdfs:label>
<rdfs:label rdf:datatype="http://www.w3.org/2001/XMLSchema#string">caffeine</rdfs:label>
```

## Class Definition

To retrieve the full class definition, a more efficient approach is to use the `xmllint` command, which we already used in previous chapters:

```
$ xmllint --xpath "//*[local-name()='label' and text()='malignant hyperthermia']/.." doid.owl
```

The XPath query starts by finding the label that contains *malignant hyperthermia* and then `..` gives the parent element, in this case the `Class` element.

From the output we can see that the semantics of *malignant hyperthermia* is much more than its label:

```

<owl:Class rdf:about="http://
 purl.obolibrary.org/obo/
 DOID_8545">
 <rdfs:subClassOf
 rdf:resource="http://
 purl.obolibrary.org/obo
 /DOID_0050736"/>
 <rdfs:subClassOf
 rdf:resource="http://
 purl.obolibrary.org/obo
 /DOID_66"/>
 <rdfs:subClassOf>
 <owl:Restriction>
 <owl:onProperty
 rdf:resource="
 http://purl.
 obolibrary.org/
 obo/IDO_0000664"/
 >
 <owl:someValuesFrom
 rdf:resource="
 http://purl.
 obolibrary.org/
 obo/GENO_0000147"
 />
 </owl:Restriction>
 </rdfs:subClassOf>
 <obo:IAO_0000115
 ...
 <oboInOwl:hasDbXref
 rdf:datatype="http://
 www.w3.org/2001/
 XMLSchema#string">
 UMLS_CUI:C0024591</
 oboInOwl:hasDbXref>
 <oboInOwl:hasExactSynonym
 rdf:datatype="http://
 www.w3.org/2001/
 XMLSchema#string">
 anesthesia related
 hyperthermia</
 oboInOwl:hasExactSynonym
 >
 <oboInOwl:hasExactSynonym
 rdf:datatype="http://
 www.w3.org/2001/
 XMLSchema#string">
 malignant hyperpyrexia
 due to anesthesia</

 oboInOwl:hasExactSynonym
 >
 <oboInOwl:hasOBONamespace
 rdf:datatype="http://
 www.w3.org/2001/
 XMLSchema#string">
 disease_ontology</
 oboInOwl:hasOBONamespace
 >
 <oboInOwl:id rdf:datatype=
 "http://www.w3.org
 /2001/XMLSchema#string"
 >DOID:8545</oboInOwl:id
 >
 <oboInOwl:inSubset
 rdf:resource="http://
 purl.obolibrary.org/obo
 /doid#DO_MGI_slim"/>
 <oboInOwl:inSubset
 rdf:resource="http://
 purl.obolibrary.org/obo
 /doid#DO_rare_slim"/>
 <oboInOwl:inSubset
 rdf:resource="http://
 purl.obolibrary.org/obo
 /doid#NCIthesaurus"/>
 <rdfs:comment rdf:datatype
 ="http://www.w3.org
 /2001/XMLSchema#string"
 >Xref MGI.
 OMIM mapping confirmed by DO. [
 SN].</rdfs:comment>
 <rdfs:label rdf:datatype="
 http://www.w3.org/2001/
 XMLSchema#string">
 malignant hyperthermia<
 /rdfs:label>
 </owl:Class>

```

A graphical visualization of this class is depicted in Fig. 5.1.

Class: malignant hyperthermia

Term IRI: [http://purl.obolibrary.org/obo/DOID\\_8545](http://purl.obolibrary.org/obo/DOID_8545)

**Definition:** A muscle tissue disease that is characterized by a drastic and uncontrolled increase in skeletal muscle oxidative metabolism, which overwhelms the body's capacity to supply oxygen, remove carbon dioxide, and regulate body temperature. [database\_cross\_reference: [url:http://en.wikipedia.org/wiki/Malignant\\_hyperthermia](http://en.wikipedia.org/wiki/Malignant_hyperthermia)][database\_cross\_reference: [url:http://en.wikipedia.org/wiki/Malignant\\_hyperthermia](http://en.wikipedia.org/wiki/Malignant_hyperthermia)][database\_cross\_reference: [url:http://en.wikipedia.org/wiki/Malignant\\_hyperthermia](http://en.wikipedia.org/wiki/Malignant_hyperthermia)][database\_cross\_reference: [url:http://en.wikipedia.org/wiki/Malignant\\_hyperthermia](http://en.wikipedia.org/wiki/Malignant_hyperthermia)]

#### Annotations

- **database\_cross\_reference:** ICD9CM:995.86; MESH:D008305; ICD10CM:T88.3; UMLS\_CUI:C0024591; ORDO:423; CSP2005:2871-4352; GARD:6964; MTHICD9\_2006:995.86; NCI:C84869; OMIM:PS145600
- **has\_exact\_synonym:** anesthesia related hyperthermia; malignant hyperpyrexia due to anesthesia
- **has\_obo\_namespace:** disease\_ontology
- **http://www.w3.org/2000/01/rdf-schema#comment:** Xref MGI. OMIM mapping confirmed by DO. [SN].
- **id:** DOID:8545
- **in\_subset:** DO MGI slim; DO rare slim; NCItthesaurus

#### Class Hierarchy

```
Thing
+ disease
+ disease of anatomical entity
+ musculoskeletal system disease
+ muscular disease
+ muscle tissue disease
- distal arthrogryposis
- rippling muscle disease 2
- rippling muscle disease 1
- myostatin-related muscle hypertrophy
- myotonia congenita
+ myopathy
- malignant hyperthermia
```

**Fig. 5.1** Class description of *malignant hyperthermia* in the Human Disease Ontology (Source: <http://www.ontobee.org/>)

For example, we can check that *malignant hyperthermia* is a subclass of (specialization) the entries 0050736 and 66. We can directly use the link<sup>4</sup> in our browser to know more about this parent disease. We will see that it represents a *muscle tissue disease*. This means that *malignant hyperthermia* is a special case of a *muscle tissue disease*.

We can do the same to retrieve the full class definition of *caffeine*:

```
$ xmllint --xpath "//*[local-name()='label' and text()='caffeine']/.." chebi_lite.owl
```

From the output we can see that the types of semantics available for *caffeine* differs from the semantics of *malignant hyperthermia*, but they still share many important properties, such as the definition of `subClassOf`:

```

<owl:Class rdf:about="http://
 purl.obolibrary.org/obo/
 CHEBI_27732">
 <rdfs:subClassOf
 rdf:resource="http://
 purl.obolibrary.org/obo/
 /CHEBI_26385"/>
 <rdfs:subClassOf
 rdf:resource="http://
 purl.obolibrary.org/obo/
 /CHEBI_27134"/>
 <rdfs:subClassOf>
 <owl:Restriction>
 <owl:onProperty
 rdf:resource="
 http://purl.
 obolibrary.org/
 obo/RO_0000087"/>
 <owl:someValuesFrom
 rdf:resource="
 http://purl.
 obolibrary.org/
 obo/CHEBI_25435"/
 >
 </owl:Restriction>
 </rdfs:subClassOf>
 ...
 <rdfs:subClassOf>
 <owl:Restriction>
 <owl:onProperty
 rdf:resource="http:
 //purl.obolibrary.
 org/obo/RO_0000087"/
 >
 <owl:someValuesFrom
 rdf:resource="
 http://purl.
 obolibrary.org/
 obo/CHEBI_85234"/
 >
 </owl:Restriction>
 </rdfs:subClassOf>
 <obo:IAO_0000115
 rdf:datatype="http://
 www.w3.org/2001/
 XMLSchema#string">A
 trimethylxanthine in
 which the three methyl
 groups are located at
 positions 1, 3, and 7.
 A purine alkaloid that
 occurs naturally in tea
 and coffee.</
 obo:IAO_0000115>

 <oboInOwl:hasAlternativeId
 rdf:datatype="http://
 www.w3.org/2001/XML
 Schema#string">CHEBI:
 22982</oboInOwl:has
 AlternativeId>
 <oboInOwl:hasAlternativeId
 rdf:datatype="http://
 www.w3.org/2001/
 XMLSchema#string">
 CHEBI:3295</oboInOwl:
 hasAlternativeId>
 <oboInOwl:hasAlternativeId
 rdf:datatype="http://
 www.w3.org/2001/XML
 Schema#string">CHEBI:
 41472</oboInOwl:
 hasAlternativeId>
 <oboInOwl:hasOBONamespace
 rdf:datatype="http://
 www.w3.org/2001/
 XMLSchema#string">
 chebi_ontology</
 oboInOwl:hasOBONamespace
 >
 <oboInOwl:id rdf:datatype=
 "http://www.w3.org
 /2001/XMLSchema#string"
 >CHEBI:27732</
 oboInOwl:id>
 <oboInOwl:inSubset
 rdf:resource="http://
 purl.obolibrary.org/obo
 /chebi#3_STAR"/>
 <rdfs:label rdf:datatype="
 http://www.w3.org/2001/
 XMLSchema#string">
 caffeine</rdfs:label>
</owl:Class>

```

A graphical visualization of this class is depicted in Fig. 5.2.

Class: caffeine

Term IRI: [http://purl.obolibrary.org/obo/CHEBI\\_27732](http://purl.obolibrary.org/obo/CHEBI_27732)

Definition: A trimethylxanthine in which the three methyl groups are located at positions 1, 3, and 7. A purine alkaloid that occurs naturally in tea and coffee.

Annotations

- **database\_cross\_reference:** PMID:15257305; PMID:10822912; PMID:18421070; PMID:16528931; PMID:22770225; PMID:12943586; PMID:17957400; PMID:8679661; PMID:12397877; KNApSAcK:C00001492; PMID:14521986; PMID:11815511; PMID:11431501; PMID:20164568; Beilstein:17705; PMID:11209966; PMID:9132918; PMID:11410911; PMID:16709440; PMID:11014293; PMID:18625110; Gmelin:103040; MetaCyc:1-3-7-TRIMETHYLXANTHINE; PMID:19879252; KEGG:C07481; PMID:12457274; PMID:10803761; PMID:19088793; HMDB:HMDB0001847; PMID:7689104; PMID:14607010; KEGG:D00528; PMID:16143823; PMID:11949272; DrugBank:DB00201; PMID:15280431; PMID:10884512; PMID:17387608; PMID:16856769; PMID:19084078; PMID:16644114; PMID:10924888; PMID:10796597; PMID:11022879; LINC:LSM-2026; PMID:10510174; PMID:16805851; PMID:8347173; PDBeChem:CFF; PMID:7441110; PMID:16391865; PMID:9218278; PMID:15840517; PMID:9067318; PMID:18258404; Drug\_Central:463; PMID:19418355; PMID:17508167; PMID:17724925; PMID:12574990; PMID:10983026; PMID:15718055; Reaxys:17705; PMID:19007524; Wikipedia:Caffeine; PMID:9063686; PMID:18647558; PMID:18068204; CAS:58-08-2; PMID:17132260; PMID:20470411; PMID:8332255; PMID:11312039; PMID:15681408; PMID:17932622; PMID:19047957; PMID:12915014
- **has\_alternative\_id:** CHEBI:22982; CHEBI:41472; CHEBI:3295
- **has\_exact\_synonym:** CAFFEINE; Caffeine; 1,3,7-trimethyl-3,7-dihydro-1H-purine-2,6-dione; caffeine
- **has\_obo\_namespace:** chebi\_ontology
- **has\_related\_synonym:** Thein; guaranine; cafeine; theine; 1-methyltheobromine; 1,3,7-trimethyl-2,6-dioxopurine; 3,7-Dihydro-1,3,7-trimethyl-1H-purin-2,6-dion; 1,3,7-trimethylxanthine; anhydrous caffeine; 1,3,7-Trimethylxanthine; 7-methyltheophylline; Coffein; cafeina; 1,3,7-trimethylpurine-2,6-dione; mateina; methyltheobromine; Koffein; teina
- **http://purl.obolibrary.org/obo/chebi/charge:** 0
- **http://purl.obolibrary.org/obo/chebi/formula:** C8H10N4O2
- **http://purl.obolibrary.org/obo/chebi/inchi:** InChI=1S/C8H10N4O2/c1-10-4-9-6-5(10)7(13)12(3)8(14)11(6)2/h4H,1-3H3
- **http://purl.obolibrary.org/obo/chebi/inchikey:** RYYVLZVUVIJVGH-UHFFFAOYSA-N
- **http://purl.obolibrary.org/obo/chebi/mass:** 194.19076
- **http://purl.obolibrary.org/obo/chebi/monoisotopicmass:** 194.080
- **http://purl.obolibrary.org/obo/chebi/smiles:** Cn1cnc2n(C)c(=O)n(C)c(=O)c21
- **http://www.geneontology.org/formats/oboInOwl#id:** CHEBI:27732
- **in\_subset:** [http://purl.obolibrary.org/obo/chebi#3\\_STAR](http://purl.obolibrary.org/obo/chebi#3_STAR)

Class Hierarchy

```
Thing
+ chemical entity
+ molecular entity
+ main group molecular entity
+ p-block molecular entity
+ carbon group molecular entity
+ organic molecular entity
+ organic molecule
+ organic cyclic compound
+ organic heterocyclic compound
+ organic heteropolycyclic compound
+ organic heterobicyclic compound
+ imidazopyrimidine
+ purines
+ purine alkaloid
+ methylxanthine
+ trimethylxanthine
- 8-(3-chlorostyryl)caffeine
- caffeine
```

Fig. 5.2 Class description of *caffeine* in ChEBI (Source: <http://www.ontobee.org/>)

The class *caffeine* is a specialization of two other entries: 26385 (*purine alkaloid*<sup>5</sup>), and 27134 (*trimethylxanthine*<sup>6</sup>). However, it contains additional subclass relationships that do not represent subsumption (*is-a*).

## Related Classes

Figures 5.3 and 5.4 show other related classes of *malignant hyperthermia* and



*caffeine*, respectively.

#### Superclasses & Asserted Axioms

- [muscle tissue disease](#)
- [autosomal dominant disease](#)
- [has material basis in some autosomal dominant inheritance](#)

**Fig. 5.3** Related classes of *malignant hyperthermia* in the Human Disease Ontology (Source: <http://www.ontobee.org/>)

#### Superclasses & Asserted Axioms

- [has role some human blood serum metabolite](#)
- [has role some mouse metabolite](#)
- [has role some plant metabolite](#)
- [has role some fungal metabolite](#)
- [has role some environmental contaminant](#)
- [has role some adjuvant](#)
- [has role some food additive](#)
- [has role some ryanodine receptor agonist](#)
- [has role some adenosine receptor antagonist](#)
- [has role some ryanodine receptor modulator](#)
- [has role some EC 3.1.4.\\* \(phosphoric diester hydrolase\) inhibitor](#)
- [has role some EC 2.7.11.1 \(non-specific serine/threonine protein kinase\) inhibitor](#)
- [has role some adenosine A2A receptor antagonist](#)
- [has role some central nervous system stimulant](#)
- [has role some psychotropic drug](#)
- [has role some diuretic](#)
- [has role some xenobiotic](#)
- [has role some mutagen](#)
- [purine alkaloid](#)
- [trimethylxanthine](#)

**Fig. 5.4** Related classes of *caffeine* in ChEBI (Source: <http://www.ontobee.org/>)

For example, the relationship between *caffeine* and the entry 25435 (*mutagen*<sup>7</sup>) is defined by the entry 0000087 (*has role*<sup>8</sup>) of the *Relations Ontology*. This means that the relationship defines that *caffeine has role mutagen*.

We can also search in the OWL file for the definition of the type of relation *has role*:

```
$ xmllint --xpath "//*[local-name()='ObjectProperty'][@*
 [local-name()='about']='http://purl.obolibrary.org/obo/RO_0000087']"
 chebi_lite.owl
```

The XPath query starts by finding the elements `ObjectProperty` and then selects the ones containing the `about` attribute with the relation URI as value.

We can check that the relation is neither transitive or cyclic:

```
<owl:ObjectProperty rdf:about="
 http://purl.obolibrary.org/
 obo/RO_0000087">
 <oboInOwl:hasDbXref
 rdf:datatype="http://
 www.w3.org/2001/
 XMLSchema#string">
 RO:0000087</
 oboInOwl:hasDbXref>
 <oboInOwl:hasOBONamespace
 rdf:datatype="http://
 www.w3.org/2001/
 XMLSchema#string">
 chebi_ontology</
 oboInOwl:hasOBONamespace
 >
 <oboInOwl:id rdf:datatype=
 "http://www.w3.org
 /2001/XMLSchema#string"
 >has_role</oboInOwl:id>
<oboInOwl:is_cyclic rdf:
 datatype="http://www.w3
 .org/2001/XMLSchema#
 boolean">false</
 oboInOwl:is_cyclic>
<oboInOwl:is_transitive
 rdf:datatype="http://
 www.w3.org/2001/
 XMLSchema#boolean">
 false</oboInOwl:
 is_transitive>
```

```

<oboInOwl:shorthand rdf:
 datatype="http://www.w3
 .org/2001/XMLSchema#
 string">has_role</
 oboInOwl:shorthand>
<rdfs:label rdf:datatype="
 http://www.w3.org/2001/
 /XMLSchema#string">has
 role</rdfs:label>
</owl:ObjectProperty>

```

A graphical visualization of this property is depicted in Fig. 5.5.

ObjectProperty: has role

Term IRI: [http://purl.obolibrary.org/obo/RO\\_0000087](http://purl.obolibrary.org/obo/RO_0000087)

Annotations

- database\_cross\_reference: RO:0000087
- has\_obo\_namespace: chebi\_ontology
- [http://www.geneontology.org/formats/oboInOwl#id:has\\_role](http://www.geneontology.org/formats/oboInOwl#id:has_role)
- [http://www.geneontology.org/formats/oboInOwl#is\\_cyclic>false](http://www.geneontology.org/formats/oboInOwl#is_cyclic>false)
- [http://www.geneontology.org/formats/oboInOwl#is\\_transitive>false](http://www.geneontology.org/formats/oboInOwl#is_transitive>false)
- shorthand: has\_role

Fig. 5.5 Description of *has role* property (Source: <http://www.ontobee.org/>)

## URIs and Labels

In the previous examples, we searched the OWL file using labels and URIs. To standardize the process, we will create two scripts that will convert a label into a URI and vice-versa. The idea is to perform all the internal ontology processing using the URIs and in the end convert them to labels, so we can use them in text processing.

### URI of a Label

To get the URI of *malignant hyperthermia*, we can use the following query:

```

$ xmllint --xpath "//*[local-
 name()='label' and text()
 ='malignant hyperthermia
 ']/../@*[local-name()='
 about']" doid.owl

```

We added the `@*[local-name()='about']` to extract the URI



specified as an attribute of that class.

The output will be the name of the attribute and its value:

```
rdf:about="http://purl.
 obolibrary.org/obo/DOID_8545"
```

To extract only the value, we can add the `string` function to the XPath query:

```
$ xmllint --xpath "string(//*[
 local-name()='label' and
 text()='malignant
 hyperthermia']/../@*[local
 -name()='about'])" doid.
 owl
```

Unfortunately, the `string` function returns only one attribute value, even if many are matched. Nonetheless, we use the `string` function because we assume that *malignant hyperthermia* is an unambiguous label, i.e. only one class will match.

The output will now be only the attribute value:

```
http://purl.obolibrary.org/obo/
 DOID_8545
```

To get the URI of *caffeine* is just about the same command:

```
$ xmllint --xpath "string(//*[
 local-name()='label' and
 text()='caffeine']/../@*[
 local-name()='about'])"
 chebi_lite.owl
```

We can now write a script that receives multiple labels given as standard input and the OWL file where to find the URIs as argument. Thus, we can create the script named *geturi.sh* with the following lines:

```
1 OWLFILE=$1
2 xargs -I {} xmllint --xpath
 "//*[local-name()='label'
 and
3 text()='{}']/../@*[local-
 name
4 ()='about']" $OWLFILE | \
5 tr ' ' '\n' | grep 'http'
```

Again we cannot forget to save the file in our working directory, and add the

right permissions using `chmod` as we did with our scripts in the previous chapters. The `xargs` command is used to process each line of the standard input. The `tr` command was added because `xmlint` displays all the matches in the same line, so we split the output using the character delimiting the URI, i.e. ". Then we use the `grep` command to keep only the lines with a URI, i.e. the ones that contain the term *http*.

Now to execute the script we only need to provide the labels as standard input:

```
$ echo 'malignant hyperthermia'
 | ./geturi.sh doid.owl
$ echo 'caffeine' | ./geturi.sh
 chebi_lite.owl
```

The output should be the URIs of those classes:

```
http://purl.obolibrary.org/obo/
 DOID_8545
http://purl.obolibrary.org/obo/
 CHEBI_27732
```

We can also execute the script using multiple labels, one per line:

```
$ echo -e 'malignant
 hyperthermia\nmuscle
 tissue disease' | ./geturi
 .sh doid.owl
$ echo -e 'caffeine\npurine
 alkaloid\
 ntrimethylxanthine' | ./
 geturi.sh chebi_lite.owl
```

The output will be a URI for each label:

```
http://purl.obolibrary.org/obo/
 DOID_8545
http://purl.obolibrary.org/obo/
 DOID_66
http://purl.obolibrary.org/obo/
 CHEBI_27732
http://purl.obolibrary.org/obo/
 CHEBI_26385
http://purl.obolibrary.org/obo/
 CHEBI_27134
```

## Label of a URI

To get the label of the disease entry with the identifier 8545, we can also use the `xmllint` command:

```
$ xmllint --xpath "//*[local-name()='Class'][@*[local-name()='about']='http://purl.obolibrary.org/obo/DOID_8545']/*[local-name()='label']/text()" doid.owl
```

We added the `@*[local-name()='label']` to select the element within the class that describes the label.

The output should be the label we were expecting:

malignant hyperthermia

We can do the same to get the label of the compound entry with the identifier 27732:

```
$ xmllint --xpath "//*[local-name()='Class'][@*[local-name()='about']='http://purl.obolibrary.org/obo/CHEBI_27732']/*[local-name()='label']/text()" chebi_lite.owl
```

Again, the output should be the label we were expecting:

caffeine

We can now write a script that receives multiple URIs given as standard input and the OWL file where to find the labels. We can create a script named *getlabels.sh* with the following lines:

```
1 OWLFILE=$1
2 xargs -I {} xmllint --xpath
 "//*[local-name()='Class'][@*[local-name()='about']
 '=' '{}']/*[local-name()='label']" $OWLFILE | \
3 tr '<>' '\n' | \
4 grep -v -e ':label' -e '^$'
```

The `xargs` command is used to process each line of the standard input. The `text` function does not add a newline character after each match, so if we have

multiple matches is almost impossible to separate them. This explains why we removed the `text` function from the XPath. Then we have to split the result in multiple lines using the `tr` command and filtering the lines that contain the `:label` keyword or are empty.

Now to execute the script we only need to provide the URIs as standard input:

```
$ echo 'http://purl.obolibrary.org/obo/DOID_8545' | ./getlabels.sh doid.owl
$ echo 'http://purl.obolibrary.org/obo/CHEBI_27732' | ./getlabels.sh chebi_lite.owl
```

The output should be the labels of those classes:

malignant hyperthermia

caffeine

We can also execute the script with multiple URIs:

```
$ echo -e 'http://purl.obolibrary.org/obo/DOID_8545\nhttp://purl.obolibrary.org/obo/DOID_66' | ./getlabels.sh doid.owl
$ echo -e 'http://purl.obolibrary.org/obo/CHEBI_27732\nhttp://purl.obolibrary.org/obo/CHEBI_26385\nhttp://purl.obolibrary.org/obo/CHEBI_27134' | ./getlabels.sh chebi_lite.owl
```

The output will be a label for each URI:

malignant hyperthermia

muscle tissue disease

```
caffeine
purine alkaloid
trimethylxanthine
```

To test both scripts, we can feed the output of one as the input of the other, for example:

```
$ echo -e 'malignant
hyperthermia\nmuscle
tissue disease' | ./geturi
.sh doid.owl | ./getlabels
.sh doid.owl
$ echo -e 'caffeine\npurine
alkaloid\
ntrimethylxanthine' | ./
geturi.sh chebi_lite.owl
| ./getlabels.sh chebi_lite.owl
```

The output will be the original input, i.e. the labels given as arguments to the echo command:

```
malignant hyperthermia
muscle tissue disease
caffeine
purine alkaloid
trimethylxanthine
```

Now we can use the URIs as input:

```
$ echo -e 'http://purl.
obolibrary.org/obo/
DOID_8545\nhttp://purl.
obolibrary.org/obo/DOID_66
' | ./getlabels.sh doid.
owl | ./geturi.sh doid.owl
$ echo -e 'http://purl.
obolibrary.org/obo/
CHEBI_27732\nhttp://purl.
obolibrary.org/obo/
CHEBI_26385\nhttp://purl.
obolibrary.org/obo/
CHEBI_27134' | ./getlabels
.sh
```

```
chebi_lite.owl | ./geturi.
sh
chebi_lite.owl
```

Again the output will be the original input, i.e. the URIs given as arguments to the `echo` command:

```
http://purl.obolibrary.org/obo/
DOID_8545
http://purl.obolibrary.org/obo/
DOID_66
http://purl.obolibrary.org/obo/
CHEBI_27732
http://purl.obolibrary.org/obo/
CHEBI_26385
http://purl.obolibrary.org/obo/
CHEBI_27134
```

---

## Synonyms

Concepts are not always mentioned using the same official label. Frequently, we can find in text alternative labels. This is why some of the classes also specify alternative labels, such as the ones represented by the element `hasExactSynonym`.

For example, to find all the synonyms of a disease, we can use the same XPath as used before but replacing the keyword `label` by `hasExactSynonym`:

```
$ xmllint --xpath "//*[local-
name()='Class'][@*[local-
name()='about']='http://
purl.obolibrary.org/obo/
DOID_8545']/*[local-name()
='hasExactSynonym']" doid.
owl
```

The output will be the two synonyms of *malignant hyperthermia*:



```

<oboInOwl:hasExactSynonym
 rdf:datatype="http://www.w3.
 org/2001/XMLSchema#string">
 anesthesia related
 hyperthermia</
 oboInOwl:hasExactSynonym>
<oboInOwl:hasExactSynonym
 rdf:datatype="http://www.w3.
 org/2001/XMLSchema#string">
 malignant hyperpyrexia due to
 anesthesia</oboInOwl:
 hasExactSynonym>

```

We can also get both the primary label and the synonyms. We only need to add an alternative match to the keyword label:

```

1 xmllint --xpath "//*[local-
 name()='Class'][@*[local-
 name()='about']='http://
 purl.obolibrary.org/obo/
 DOID_8545']/*[local-name()
 ='hasExactSynonym' or
 local-name()='label']"
 doid.owl

```

The output will include now the two synonyms plus the official label:

```

<oboInOwl:hasExactSynonym
 rdf:datatype="http://www.w3.
 org/2001/XMLSchema#string">
 anesthesia related
 hyperthermia</
 oboInOwl:hasExactSynonym>
<oboInOwl:hasExactSynonym
 rdf:datatype="http://www.w3.
 org/2001/XMLSchema#string">
 malignant hyperpyrexia due to
 anesthesia</
 oboInOwl:hasExactSynonym>
<rdfs:label rdf:datatype="http:
 //www.w3.org/2001/XMLSchema#
 string">malignant
 hyperthermia</rdfs:label>

```

Thus, we can now update the script *getlabels.sh* to include synonyms:

```

1 OWLFILE=$1
2 xargs -I {} xmllint --xpath
 "//*[local-name()='Class
 '][@*[local-name()='about
 ']='{}']/*[local-name()='
 hasExactSynonym' or local-
 name()='hasRelatedSynonym'
 or local-name()='label']"
 $OWLFILE | \
3 tr '<>' '\n' | \
4 grep -v -e ':label' -e ':
 hasExactSynonym' -e '
 hasRelatedSynonym' -e '^$'

```

We should note that the XPath query and the grep command were modified by adding the `hasExactSynonym` keyword. We also added the `hasRelatedSynonym` which is available for some classes.

We can test the script exactly in the same way as before:



```
$ echo -e 'http://purl.obolibrary.org/obo/DOID_8545' | ./getlabels.sh doid.owl
```

But now the output will display multiple labels for this class:

```
anesthesia related hyperthermia
malignant hyperpyrexia due to
 anesthesia
malignant hyperthermia
```

## URI of Synonyms

Since the script now returns alternative labels, we may encounter some problems if we send the output to the *geturi.sh* script:

```
$ echo 'http://purl.obolibrary.org/obo/DOID_8545' | ./getlabels.sh doid.owl | ./geturi.sh doid.owl
```

The previous command will display XPath warnings for the two synonyms:

```
XPath set is empty
XPath set is empty
http://purl.obolibrary.org/obo/DOID_8545
```

If we do not want to know about these mismatches, we can always redirect them to the null device:

```
$ echo 'http://purl.obolibrary.org/obo/DOID_8545' | ./getlabels.sh doid.owl | ./geturi.sh doid.owl 2>/dev/null
```

However, we can update the script *geturi.sh* to also include synonyms:

```

1 OWLFILE=$1
2 xargs -I {} xmllint --xpath
 "//*[(local-name()='
 hasExactSynonym' or local-
 name()='hasRelatedSynonym'
 or local-name()='label')
 and text()='{}']/../@*[
 local-name()='about']"
 $OWLFILE | \
3 tr '"' '\n' | grep 'http'

```

Now we can execute the same command:

```

$ echo 'http://purl.obolibrary.
 org/obo/DOID_8545' | ./
 getlabels.sh doid.owl | ./
 geturi.sh doid.owl

```

Every label should now be matched exactly with the same class:

```

http://purl.obolibrary.org/obo/
 DOID_8545
http://purl.obolibrary.org/obo/
 DOID_8545
http://purl.obolibrary.org/obo/
 DOID_8545

```

If we want to avoid repetitions, we can add the `sort` command with the `-u` option to the end of each command, as we did in previous chapters:

```

$ echo 'http://purl.obolibrary.
 org/obo/DOID_8545' | ./
 getlabels.sh doid.owl | ./
 geturi.sh doid.owl | sort
 -u

```

The output should now be only one URI:

```

http://purl.obolibrary.org/obo/
 DOID_8545

```

---

## Parent Classes

Parent classes represent generalizations that may also be relevant to recognize in text. To extract all the parent classes of *malignant hyperthermia*, we can use the

following XPath query:

```
$ xmllint --xpath "//*[local-
 name()='Class'][@*[local-
 name()='about']='http://
 purl.obolibrary.org/obo/
 DOID_8545']/*[local-name()
 ='subClassOf']/@*[local-
 name()='resource']" doid.
owl
```

The first part of the XPath is the same as the above to get the class element, then `[local-name()='subClassOf']` is used to get the subclass element, and finally `@*[local-name()='resource']` is used to get the attribute containing its URI.

The output should be the URIs representing the parents of class 8545:

```
rdf:resource="http://purl.
 obolibrary.org/obo/
 DOID_0050736"
rdf:resource="http://purl.
 obolibrary.org/obo/DOID_66"
```

We can also execute the same command for *caffeine*:

```
$ xmllint --xpath "//*[local-
 name()='Class'][@*[local-
 name()='about']='http://
 purl.obolibrary.org/obo/
 CHEBI_27732']/*[local-name
 ()='subClassOf']/@*[local-
 name()='resource']"
chebi_lite.owl
```

The output will now include two parents:

```
rdf:resource="http://purl.
 obolibrary.org/obo/
 CHEBI_26385"
rdf:resource="http://purl.
 obolibrary.org/obo/
 CHEBI_27134"
```

We should note that we no longer can use the `string` function, because

ontologies are organized as DAGs using multiple inheritance, i.e. each class can have multiple parents, and the `string` function only returns the first match. To get only the URIs, we can apply the previous technique of using the `tr` and `grep` commands:

```
$ xmllint --xpath "//*[local-name()='Class']/@*[local-name()='about']='http://purl.obolibrary.org/obo/CHEBI_27732'//*[@local-name()='subClassOf']/@*[local-name()='resource']" chebi_lite.owl | tr '"' '\n' | grep 'http'
```

Now the output only contains the URIs:

```
http://purl.obolibrary.org/obo/CHEBI_26385
http://purl.obolibrary.org/obo/CHEBI_27134
```

We can now create a script that receives multiple URIs given as standard input and the OWL file where to find all the parents as argument. The script named *getparents.sh* should contain the following lines:

```
1 OWLFILE=$1
2 xargs -I {} xmllint --xpath
 "//*[local-name()='Class']/@*[local-name()='about']='{}'//*[@local-name()='subClassOf']/@*[local-name()='resource']" $OWLFILE |
 \
3 tr '"' '\n' | grep 'http'
```

To get the parents of *malignant hyperthermia*, we will only need to give the URI as input and the OWL file as argument:

```
$ echo 'http://purl.obolibrary.org/obo/DOID_8545' | ./getparents.sh doid.owl
```

The output will include the URIs of the two parents:

```
http://purl.obolibrary.org/obo/
 DOID_0050736
http://purl.obolibrary.org/obo/
 DOID_66
```

## Labels of Parents

But if we need the labels we can redirect the output to the *getlabels.sh* script:

```
$ echo 'http://purl.obolibrary.
 org/obo/DOID_8545' | ./
 getparents.sh doid.owl |
 ./getlabels.sh doid.owl
```

The output will now be the label of the parents of *malignant hyperthermia*:  
autosomal dominant disease

muscle tissue disease

Again, the same can be done with *caffeine*:

```
$ echo 'http://purl.obolibrary.
 org/obo/CHEBI_27732' | ./
 getparents.sh chebi_lite.
 owl | ./getlabels.sh
 chebi_lite.owl
```

And now the output contains the labels of the parents of *caffeine*:

purine alkaloid

trimethylxanthine

## Related Classes

If we are interested in using all the related classes besides the ones that represent a generalization (*subClassOf*), we have to change our XPath to:

```
$ xmllint --xpath "//*[local-
 name()='Class'][@*[local-
 name()='about']='http://
 purl.obolibrary.org/obo/
 CHEBI_27732']/*[local-name
 ()='subClassOf']//*[local-
 name()='someValuesFrom']/@
 *[local-name()='resource
 ']" chebi_lite.owl | tr
 '"' '\n'| grep 'http'
```

We should note that these related classes are in the attribute *resource* of *someValuesFrom* element inside a *subClassOf* element.

The URIs of the 18 related classes of *caffeine* are now displayed:

[http://purl.obolibrary.org/obo/  
CHEBI\\_25435](http://purl.obolibrary.org/obo/CHEBI_25435)

[http://purl.obolibrary.org/obo/  
CHEBI\\_35337](http://purl.obolibrary.org/obo/CHEBI_35337)

[http://purl.obolibrary.org/obo/  
CHEBI\\_35471](http://purl.obolibrary.org/obo/CHEBI_35471)

[http://purl.obolibrary.org/obo/  
CHEBI\\_35498](http://purl.obolibrary.org/obo/CHEBI_35498)

[http://purl.obolibrary.org/obo/  
CHEBI\\_35703](http://purl.obolibrary.org/obo/CHEBI_35703)

[http://purl.obolibrary.org/obo/  
CHEBI\\_38809](http://purl.obolibrary.org/obo/CHEBI_38809)

[http://purl.obolibrary.org/obo/  
CHEBI\\_50218](http://purl.obolibrary.org/obo/CHEBI_50218)

[http://purl.obolibrary.org/obo/  
CHEBI\\_50925](http://purl.obolibrary.org/obo/CHEBI_50925)

[http://purl.obolibrary.org/obo/  
CHEBI\\_53121](http://purl.obolibrary.org/obo/CHEBI_53121)

[http://purl.obolibrary.org/obo/  
CHEBI\\_60809](http://purl.obolibrary.org/obo/CHEBI_60809)

[http://purl.obolibrary.org/obo/  
CHEBI\\_64047](http://purl.obolibrary.org/obo/CHEBI_64047)

[http://purl.obolibrary.org/obo/  
CHEBI\\_67114](http://purl.obolibrary.org/obo/CHEBI_67114)

[http://purl.obolibrary.org/obo/  
CHEBI\\_71232](http://purl.obolibrary.org/obo/CHEBI_71232)

[http://purl.obolibrary.org/obo/  
CHEBI\\_75771](http://purl.obolibrary.org/obo/CHEBI_75771)

[http://purl.obolibrary.org/obo/  
CHEBI\\_76924](http://purl.obolibrary.org/obo/CHEBI_76924)

[http://purl.obolibrary.org/obo/  
CHEBI\\_76946](http://purl.obolibrary.org/obo/CHEBI_76946)

[http://purl.obolibrary.org/obo/  
CHEBI\\_78298](http://purl.obolibrary.org/obo/CHEBI_78298)

[http://purl.obolibrary.org/obo/  
CHEBI\\_85234](http://purl.obolibrary.org/obo/CHEBI_85234)



## Labels of Related Classes

To get the labels of these related classes, we only need to add the *getlabels.sh* script:

```
$ xmllint --xpath "//*[local-name()='Class'][@*[local-name()='about']='http://purl.obolibrary.org/obo/CHEBI_27732']/*[local-name()='subClassOf']//*[local-name()='someValuesFrom']/@*[local-name()='resource']" chebi_lite.owl | tr
''' '\n'| grep 'http' | ./
getlabels.sh chebi_lite.
owl
```

The output is now 18 terms that we could use to expand our text processing:



mutagen  
central nervous system stimulant  
psychotropic drug  
diuretic  
xenobiotic  
ryanodine receptor modulator  
EC 3.1.4.\* (phosphoric diester  
hydrolase) inhibitor  
EC 2.7.11.1 (non-specific serine  
/threonine protein kinase)  
inhibitor  
adenosine A2A receptor  
antagonist  
adjuvant  
food additive  
ryanodine receptor agonist  
adenosine receptor antagonist  
mouse metabolite  
plant metabolite  
fungal metabolite  
environmental contaminant  
human blood serum metabolite

---

## Ancestors

Finding all the ancestors of a class includes many chain invocations of the *getparents.sh* until we get no matches. We also should avoid relations that are cyclic, otherwise we will enter in a infinite loop. Thus, for identifying the ancestors of a class, we will only consider parent relations, i.e. subsumption relations.

## Grandparents

In the previous section we were able to extract the direct parents of a class, but the parents of these parents also represent generalizations of the original class. For example, to get the parents of the parents (grandparents) of *malignant hyperthermia* we need to invoke *getparents.sh* twice:

```
$ echo 'malignant hyperthermia'
 | ./geturi.sh doid.owl |
 ./getparents.sh doid.owl |
 ./getparents.sh doid.owl
```

And we will find the URIs of the grandparents of *malignant hyperthermia*:

```
http://purl.obolibrary.org/obo/
DOID_0050739
http://purl.obolibrary.org/obo/
DOID_0080000
```

Or to get their labels we can add the *getlabels.sh* script:

```
$ echo 'malignant hyperthermia'
 | ./geturi.sh doid.owl |
 ./getparents.sh doid.owl |
 ./getparents.sh doid.owl
 | ./getlabels.sh doid.owl
```

And we find the labels of the grandparents of *malignant hyperthermia*:

```
autosomal genetic disease
muscular disease
```

## Root Class

However, there are classes that do not have any parent, which are called root classes. In Figs. 5.1 and 5.2, we can see that *disease* and *chemical entity* are root classes of DO and ChEBI ontologies, respectively. As we can see these are highly generic terms.

To check if it is the root class, we can ask for their parents:

```
$ echo 'disease' | ./geturi.sh
 doid.owl | ./getparents.sh
 doid.owl
$ echo 'chemical entity' | ./
 geturi.sh chebi_lite.owl |
 ./getparents.sh
 chebi_lite.owl
```

In both cases, we will get the warning that no matches were found, confirming that they are the root class.

```
XPath set is empty
```

## Recursion

We can now build a script that receives a list of URIs as standard input, and

invokes *getparents.sh* recursively until it reaches the root class.

The script named *getancestors.sh* should contain the following lines:

```
1 OWLFILE=$1
2 CLASSES=$(cat -)
3 [[-z "$CLASSES"]] && exit
4 PARENTS=$(echo "$CLASSES" | ./
 getparents.sh $OWLFILE |
 sort -u)
5 echo "$PARENTS"
6 echo "$PARENTS" | ./
 getancestors.sh $OWLFILE
```

The second line of the script saves the standard input in a variable named `CLASSES`, because we need to use it twice: (i) to check if the input as any classes or is empty (third line) and (ii) to get the parents of the classes given as input (fourth line). If the input is empty then the script ends, this is the base case of the recursion<sup>9</sup>. This is required so the recursion stops at a given point. Otherwise, the script would run indefinitely until the user stops it manually.

The fourth line of the script stores the output in a variable named `PARENTS`, because we need also to use it twice: (i) to output these direct parents (fifth line), and (ii) to get the ancestors of this parents (sixth line). We should note that we are invoking the *getancestors.sh* script inside the *getancestors.sh*, which defines the recursion step. Since the subsumption relation is acyclic, we expect that at some time we will reach classes without parents (root classes) and then the script will end.

We should note that the `echo` of the variables `CLASSES` and `PARENTS` need to be inside commas, so the newline characters are preserved.

## Iteration

Recursion is most of the times computational expensive, but usually it is possible to replace recursion with iteration to develop a more efficient algorithm. Explaining iteration and how to refactor a recursive script is out of scope of this book, nevertheless the following script represents an equivalent way to get all the ancestors without using recursion:

```

1 # iteration
2 OWLFILE=$1
3 CLASSES=$(cat -)
4 ANCESTORS=""
5 while [[! -z "$CLASSES"]]
6 do
7 PARENTS=$(echo "$CLASSES" |
 ./getparents.sh $OWLFILE
 | sort -u)
8 ANCESTORS="$ANCESTORS\
 n$PARENTS"
9 CLASSES=$PARENTS
10 done
11 echo -e "$ANCESTORS"

```

The script uses the `while` command that basically implements iteration by repeating a set of commands (lines 6–8) while a given condition is satisfied (line 4).

To test the recursive script, we can provide as standard input the label *malignant hyperthermia*:

```

$ echo 'http://purl.obolibrary.
 org/obo/DOID_8545' | ./
 getancestors.sh doid.owl

```

The output will be the URIs of all its ancestors:

```
http://purl.obolibrary.org/obo/
 DOID_0050736
http://purl.obolibrary.org/obo/
 DOID_66
http://purl.obolibrary.org/obo/
 DOID_0050739
http://purl.obolibrary.org/obo/
 DOID_0080000
http://purl.obolibrary.org/obo/
 DOID_0050177
http://purl.obolibrary.org/obo/
 DOID_17
http://purl.obolibrary.org/obo/
 DOID_630
http://purl.obolibrary.org/obo/
 DOID_7
http://purl.obolibrary.org/obo/
 DOID_4
```

We should note that we will still receive the XPath warning when the script reaches the root class and no parents are found:

```
XPath set is empty
```

To remove this warning and just get the labels of the ancestors of *malignant hyperthermia*, we can redirect the warnings to the null device:

```
$ echo 'malignant hyperthermia'
 | ./geturi.sh doid.owl |
 ./getancestors.sh doid.owl
 2>/dev/null | ./getlabels
 .sh doid.owl
```

The output will now include the name of all ancestors of *malignant hyperthermia*:

```
autosomal dominant disease
muscle tissue disease
autosomal genetic disease
muscular disease
monogenic disease
musculoskeletal system disease
genetic disease
disease of anatomical entity
disease
```

We should note that the first two ancestors are the direct parents of *malignant hyperthermia*, and the last one is the root class. This happens because the recursive script print the parents before invoking itself to find the ancestors of the direct parents.

We can do the same with *caffeine*, but be advised that given the higher number of ancestors in ChEBI we may now have to wait a little longer for the script to end.

```
$ echo 'caffeine' | ./geturi.sh
 chebi_lite.owl | ./
 getancestors.sh chebi_lite
 .owl | ./getlabels.sh
 chebi_lite.owl | sort -u
```

The results include repeated classes that were found by using different branches, so that is why we need to add the `sort` command with the `-u` option to eliminate the duplicates.

The script will print the ancestors being found by the script:

alkaloid  
aromatic compound  
bicyclic compound  
carbon group molecular entity  
chemical entity  
cyclic compound  
heteroarene  
heterobicyclic compound  
heterocyclic compound  
heteroorganic entity  
heteropolycyclic compound  
imidazopyrimidine  
main group molecular entity  
methylxanthine  
molecular entity  
molecule  
nitrogen molecular entity  
organic aromatic compound  
organic cyclic compound  
organic heterobicyclic compound  
organic heterocyclic compound  
organic heteropolycyclic  
compound  
organic molecular entity  
organic molecule  
organonitrogen compound  
organonitrogen heterocyclic  
compound  
p-block molecular entity  
pnictogen molecular entity  
polyatomic entity  
polycyclic compound  
purine alkaloid  
purines  
trimethylxanthine

---

## My Lexicon

Now that we know how to extract all the labels and related classes from an ontology, we can construct our own lexicon with the list of terms that we want to recognize in text.

Let us start by creating the file *do\_8545\_lexicon.txt* representing our lexicon for *malignant hyperthermia* with all its labels:

```
$ echo 'malignant hyperthermia'
 | ./geturi.sh doid.owl |
 ./getlabels.sh doid.owl >
 do_8545_lexicon.txt
```

## Ancestors Labels

Now we can add to the lexicon all the labels of the ancestors of *malignant hyperthermia* by adding the redirection operator:

```
$ echo 'malignant hyperthermia'
 | ./geturi.sh doid.owl |
 ./getancestors.sh doid.owl
 | ./getlabels.sh doid.owl
 >> do_8545_lexicon.txt
```

We should note that now we use `>>` and not `>`, this will append more lines to the file instead of creating a new file from scratch.

Now we can check the contents of the file *do\_8545\_lexicon.txt* to see the terms we got:

```
$ cat do_8545_lexicon.txt | sort
-u
```

We should note that we use the `sort` command with the `-u` option to eliminate any duplicates that may exist.

We should be able to see the following labels:



anesthesia related hyperthermia  
autosomal dominant disease  
autosomal genetic disease  
disease  
disease of anatomical entity  
genetic disease  
malignant hyperpyrexia due to  
    anesthesia  
malignant hyperthermia  
monogenic disease  
muscle tissue disease  
muscular disease  
musculoskeletal system disease

We can also apply the same commands for *caffeine* to produce its lexicon in the file *chebi\_27732\_lexicon.txt* by adding the redirection operator:

```
$ echo 'caffeine' | ./geturi.sh
 chebi_lite.owl | ./
 getlabels.sh chebi_lite.
 owl > chebi_27732_lexicon.
 txt

$ echo 'caffeine' | ./geturi.sh
 chebi_lite.owl | ./
 getancestors.sh chebi_lite
 .owl | ./getlabels.sh
 chebi_lite.owl >>
 chebi_27732_lexicon.txt
```

We should note that it may take a while until it gets all labels.

Now let us check the contents of this new lexicon:

```
$ cat chebi_27732_lexicon.txt |
 sort -u
```

Now we should be able to see that this lexicon is much larger:

alkaloid  
aromatic compound  
bicyclic compound  
caffeine  
...

## Merging Labels

If we are interested in finding everything related to *caffeine* or *malignant hyperthermia*, we may be interested in merging the two lexicons in a file named *lexicon.txt*:

```
$ cat do_8545_lexicon.txt
 chebi_27732_lexicon.txt |
 sort -u > lexicon.txt
```

Using this new lexicon, we can recognize any mention in our previous file named *chebi\_27732\_sentences.txt*:

```
$ grep -w -i -F -f lexicon.txt
 chebi_27732_sentences.txt
```

We added the `-F` option because our lexicon is a list of fixed strings, i.e. does not include regular expressions. The equivalent long form to the `-F` option is `--fixed-strings`.

We now get more sentences, including some that do not include a direct mention to *caffeine* or *malignant hyperthermia*. For example, the following sentence was selected because it mentions *molecule*, which is an ancestor of *caffeine*:

```
The remainder of the molecule is
hydrophilic and presumably
constitutes the cytoplasmic
domain of the protein.
```

Another example is the following sentence, which was selected because it mentions *disease*, which is an ancestor of *malignant hyperthermia*:

```
Our data suggest that divergent
activity profiles may cause
varied disease phenotypes by
specific mutations.
```

We can also use our script *getentities.sh* giving this lexicon as argument. However, since we are not using any regular expressions it would be better to add the `-F` option to the `grep` command in the script, so the lexicon is interpreted as list of fixed strings to be matched. Only then we can execute the script safely:

```
$./getentities.sh lexicon.txt <
 chebi_27732_sentences.txt
```

## Ancestors Matched

Besides these two previous examples, we can check if there other ancestors

being matched by using the `grep` command with the `-o` option:

```
$ grep -o -w -F -f lexicon.txt
 chebi_27732_sentences.txt
 | sort -u
```

We can see that besides the terms *caffeine* and *malignant hyperthermia*, only one ancestor of each one of them was matched, *molecule* and *disease*, respectively:

```
caffeine
disease
malignant hyperthermia
molecule
```

This can be explained because our text is somehow limited and because we are using the official labels and we may be missing acronyms, and simple variations such as the plural of a term. To cope with this issue, we may use a stemmer<sup>10</sup>, or use all the ancestors besides subsumption. However, if our lexicon is small is better to do it manually and maybe add some regular expressions to deal with some of the variations.

---

## Generic Lexicon

Instead of using a customized and limited lexicon, we may be interested in recognizing any of the diseases represented in the ontology. By recognizing all the diseases in our *caffeine* related text, we will be able to find all the diseases that may be related to *caffeine*

### All Labels

To extract all the labels from the disease ontology we can use the same XPath query used before, but now without restricting it to any URI:

```
$ xmllint --xpath "//*[local-
 name()='Class']/*[local-
 name()='hasExactSynonym'
 or local-name()='
 hasRelatedSynonym' or
 local-name()='label']"
 doid.owl
```

We can create a script named *getalllabels.sh*, that receives as argument the OWL file where to find all labels containing the following lines:

```

1 OWLFILE=$1
2 xmllint --xpath "//*[local-
 name()='Class']/*[local-
 name()='hasExactSynonym'
 or local-name()='
 hasRelatedSynonym' or
 local-name()='label']"
 $OWLFILE | \
3 tr '<>' '\n' | \
4 grep -v -e ':label' -e ':
 hasExactSynonym' -e '
 hasRelatedSynonym' -e '^$'
 | \
5 sort -u

```

We should note that this script is similar to the *getlabels.sh* script without the *xargs*, since it does not receive a list of URIs as standard input.

Now we can execute the script to extract all labels from the OWL file:

```
$./getalllabels.sh doid.owl
```

The output will contain the full list of diseases:

```

11-beta-hydroxysteroid
 dehydrogenase deficiency type
 2
11p partial monosomy syndrome
1,4-phenylenediamine allergic
 contact dermatitis
...
Zoophilia
Zoophobia
zygomycosis

```

To create the generic lexicon, we can redirect the output to the file *diseases.txt*:

```
$./getalllabels.sh doid.owl >
 diseases.txt
```

We can check how many labels we got by using the *wc* command:

```
$ wc -l diseases.txt
```

The lexicon contains more than 29 thousand labels.

We can now recognize the lexicon entries in the sentences of the file

*chebi\_27732\_sentences.txt* by using the `grep` command:

```
$ grep -n -w -E -f diseases.txt
 chebi_27732_sentences.txt
```

However, we will get the following error:

```
grep: Unmatched) or $
```

This error happens because our lexicon contains some special characters also used by regular expressions, such as the parentheses.

One way to address this issue is to replace the `-E` option by the `-F` option, that treats each lexicon entry as a fixed string to be recognized:

```
$ grep -n -o -w -F -f diseases.
 txt chebi_27732_sentences.
 txt
```

The output will show the large list of sentences mentioning diseases:

```
1:malignant hyperthermia
2:malignant hyperthermia
9:central core disease
10:disease
10:myopathy

1092:malignant hyperthermia
1092:central core disease
1103:malignant hyperthermia
1104:malignant hyperthermia
1106:central core disease
1106:myopathy
```

## Problematic Entries

Despite using the `-F` option, the lexicon contains some problematic entries. Some entries have expressions enclosed by parentheses or brackets, that represent alternatives or a category:

```
Post measles encephalitis (
 disorder)
Glaucomatous atrophy [cupping]
 of optic disc
```

Other entries have separation characters, such as commas or colons, to represent a specialization. For example:

```
Tapeworm infection: intestinal
taenia solum
```

```
Tapeworm infection: pork
```

```
Pemphigus, Benign Familial
```

```
ATR, nondeletion type
```

A problem is that not all have the same meaning. A comma may also be part of the term. For example:

```
46,XY DSD due to LHB deficiency
```

Other case includes using *&amp;* to represent an ampersand. For example:

```
Gonococcal synovitis &/or
tenosynovitis
```

However, most of the times the alternatives are already included in the lexicon in different lines. For example:

```
Gonococcal synovitis and
tenosynovitis
Gonococcal synovitis or
tenosynovitis
```

As we can see by these examples, it is not trivial to devise rules that fully solve these issues. Very likely there will be exceptions to any rule we devise and that we are not aware of.

## Special Characters Frequency

To check the impact of each of these issues, we can count the number of times they appear in the lexicon:

```
$ grep -c -F '(' diseases.txt
$ grep -c -F ',' diseases.txt
$ grep -c -F '[' diseases.txt
$ grep -c -F ':' diseases.txt
$ grep -c -F '&' diseases.
txt
```

We will be able to see that parentheses and commas are the most frequent, with more than one thousand entries.

## Completeness

Now let us check if the *ATR* acronym representing the *alpha thalassemia-X-linked intellectual disability syndrome* is in the lexicon:

```
$ grep -E '^ATR' diseases.txt
```

All the entries include more terms than only the acronym:



```
ATR-16 syndrome
ATR, nondeletion type
ATR syndrome, deletion type
ATR syndrome linked to
 chromosome 16
ATR-X syndrome
```

Thus, a single *ATR* mention will not be recognized.

This is problematic if we need to match sentences mentioning that acronym, such as:

```
$ echo 'The ATR syndrome is an
 alpha thalassemia that has
 material basis in
 mutation in the ATRX gene
 on Xq21' | grep -w 'ATR'
```

We will now try to mitigate these issues as simply as we can. We will not try to solve them completely, but at least address the most obvious cases.

## Removing Special Characters

The first fix we will do, is to remove all the parentheses and brackets by using the `tr` command, since they will not be found in the text:

```
$ tr -d '[](){}' < diseases.txt
```

Of course, we may lose the shorter labels, such as *Post measles encephalitis*, but at least now, the disease *Post measles encephalitis disorder* will be recognized:

```
$ tr -d '[](){}' < diseases.txt
 | grep 'Post measles
 encephalitis disorder'
```

If we really need these alternatives, we would have to create multiple entries in the lexicon or transform the labels in regular expressions.

## Removing Extra Terms

The second fix is to remove all the text after a separation character, by using the `sed` command:

```
$ tr -d '[](){}' < diseases.txt
 | sed -E 's/[,:;] .*$//'
```

We should note that the regular expression enforces a space after the separation character to avoid separation characters that are not really separating two expressions, such as: *46,XY DSD due to LHB deficiency*

We can see that now we are able to recognize both *ATR* and *ATR syndrome*:

```
$ tr -d '[](){}' < diseases.txt
 | sed -E 's/[,;:] .*$//' |
 grep -E '^ATR'
```

## Removing Extra Spaces

The third fix is to remove any leading or trailing spaces of a label:

```
$ tr -d '[](){}' < diseases.txt
 | sed -E 's/[,;:] .*$//; s/^
 /*//; s/ .*$//'
```

We should note that we added two more replacement expressions to the `sed` command by separating them with a semicolon.

We can now update the script `getalllabels.sh` to include the previous `tr` and `sed` commands:

```
1 OWLFILE=$1
2 xmllint --xpath "//*[local-
 name()='Class']/*[local-
 name()=
3 'hasExactSynonym' or local-
4 name()='hasRelatedSynonym'
 or
5 local-name()='label']"
6 $OWLFILE | \
7 tr '<>' '\n' | \
8 grep -v -e ':label' -e ':
 hasExactSynonym' -e '
 hasRelatedSynonym' -e '^$'
 | \
9 tr -d '[](){}' | \
10 sed -E 's/[,;:] .*$//; s/^
 /*//; s/ .*$//' | sort -u
```

And we can now generate a fixed lexicon:

```
$./getalllabels.sh doid.owl >
 diseases.txt
```

We can check again the number of entries:

```
$ wc -l diseases.txt
```

We now have a lexicon with about 28 thousand labels. We have less entries



because our fixes made some entries equal to others already in the lexicon, and thus the `-u` option filtered them.

## Disease Recognition

We can now try to recognize lexicon entries in the sentences of file *chebi\_27732\_sentences.txt*:

```
$ grep -n -o -w -F -f diseases.
txt chebi_27732_sentences.
txt
```

To obtain the list of labels that were recognized, we can use the `grep` command:

```
$ grep -o -w -F -f diseases.txt
chebi_27732_sentences.txt
| sort -u
```

We will get a list of 43 unique labels representing diseases that may be related to *caffeine*:

Andersen-Tawil syndrome  
arrhythmogenic right ventricular  
cardiomyopathy  
ARVD2  
ataxia telangiectasia  
ATR  
atrial fibrillation  
benign congenital myopathy  
cancer  
cardiac arrest  
cardiomyopathy  
catecholaminergic polymorphic  
ventricular tachycardia  
central core disease  
chorea  
congenital hip dislocation  
congenital myopathy  
deficiency  
disease  
dystonia  
epilepsy  
FHL1  
hand  
hepatitis C  
HL  
hypercholesterolaemia  
hypokalemic periodic paralysis  
Hypokalemic periodic paralysis  
intellectual disability  
long QT syndrome  
LQT1  
LQT2  
LQT3  
LQT5  
LQT6  
malignant hyperthermia  
migraine  
myopathy  
myotonic dystrophy type 1  
nemaline myopathy  
nemaline rod myopathy  
ophthalmoplegia  
rod myopathy  
scoliosis  
syndrome

---

## Performance

The `grep` is quite efficient but of course when using large lexicons and texts we may start to feel some performing issues. Its execution time is proportional to the size of the lexicon, since each term of the lexicon will correspond to an independent pattern to match. This means that for large lexicons we may face serious performance issues.

## Inverted Recognition

A solution for dealing with large lexicons is to use the inverted recognition technique (Couto et al. 2017; Couto and Lamurias 2018). The inverted recognition uses the words of the input text as patterns to be matched against the lexicon file. When the number of words in the input text is much smaller than the number of terms in the lexicon, `grep` has much fewer patterns to match. For example, the inverted recognition technique applied to ChEBI has shown to be more than 100 times faster than using the standard technique.

## Case Insensitive

Another performance issue arises when we use the `-i` option to perform a case insensitive matching. For instance, in most computers if we execute the following command, we will have to wait much longer than not using the `-i` option:

```
$ grep -n -o -w -F -i -f
 diseases.txt
 chebi_27732_sentences.txt
```

One solution is to convert both the lexicon and text to lowercase (or uppercase), but this may result in more incorrect matches, such as incorrectly matching acronyms in lowercase.

## ASCII Encoding

The low performance issue of case insensitive matching is normally due to the usage of UTF-8 character encoding<sup>11</sup>, instead of ASCII character encoding<sup>12</sup>. UTF-8 allow us to use special characters, such as the euro symbol, in a standard way so it is interpreted by every computer around the world in the same way. However, for normal text without special characters ASCII works fine and more efficiently. In Unix shells we can normally specify the usage of ASCII encoding by adding the expression `LC\_ALL=C` before the command (`man locale` for

more information).

So, another solution is to execute the following command:

```
$ LC_ALL=C grep -n -o -w -F -i -
 f diseases.txt
 chebi_27732_sentences.txt
```

We will be able to watch the significant increase in performance.

To check how many labels are now being recognized we can execute:

```
$ LC_ALL=C grep -o -w -F -i -f
 diseases.txt
 chebi_27732_sentences.txt
 | sort -u | wc -l
```

We have now 60 labels being recognized.

To check which new labels were recognized, we can compare the results with and without the `-i` option:

```
$ LC_ALL=C grep -o -w -F -i -f
 diseases.txt
 chebi_27732_sentences.txt
 | sort -u >
 diseases_recognized_ignorecase
 .txt

$ grep -o -w -F -f diseases.txt
 chebi_27732_sentences.txt
 | sort -u >
 diseases_recognized.txt

$ grep -v -F -f
 diseases_recognized.txt
 diseases_recognized_
 ignorecase.txt
```

We are now able to see that the new labels are:

Arrhythmogenic right ventricular  
dysplasia  
arthrogryposis  
can  
Catecholaminergic polymorphic  
ventricular tachycardia  
Central Core Disease  
defect  
Disease  
dyskinesia  
face  
fever  
Malignant hyperthermia  
Malignant Hyperthermia  
March  
ORF  
total

## Correct Matches

Some important diseases could only be recognized by performing a case insensitive match, such as *arthrogryposis*. This disease was missing because in the lexicon we had the uppercase case version of the labels, but not the lowercase version. We can check it by using the `grep` command:

```
$ grep -i '^arthrogryposis$'
diseases.txt
```

The output does not include the lowercase case version:

```
Arthrogryposis
ARTHROGRYPOSIS
```

We can also check in the text which versions are used:

```
$ grep -w -i 'arthrogryposis'
chebi_27732_sentences.txt
```

We can see that only the lowercase version is used:

```
... (multiple arthrogryposis,
 congenital dislocation of the
 hips ...
... fetal akinesia,
 arthrogryposis multiplex ...
```

Another example is *dyskinesia*:

```
$ grep -i '^dyskinesia$'
diseases.txt
```

The lexicon has only the disease name with the first character in uppercase:  
Dyskinesia

## Incorrect Matches

However, using a case insensitive match may also create other problems, such as the acronym *CAN* for the disease *Crouzon syndrome-acanthosis nigricans syndrome*:

```
$ grep -i '^CAN$' diseases.txt
```

By using a case insensitive grep we will recognize the common word *CAN* as a disease. For example, we can check how many times *CAN* is recognized:

```
$ LC_ALL=C grep -n -o -w -i -F -
f diseases.txt
chebi_27732_sentences.txt
| grep -i ':CAN' | wc -l
```

It is recognized 18 times.

And to see which type of matches they are, we can execute the following command:

```
$ LC_ALL=C grep -o -w -i -F -f
diseases.txt
chebi_27732_sentences.txt
| grep -i -E '^CAN$' |
sort -u
```

We can verify that the matches are incorrect mentions of the disease acronym:

can

This means we created at least 18 mismatches by performing a case insensitive match.

---

## Entity Linking

When we are using a generic lexicon, we may be interested in identifying what the recognized labels represent. For example, we may not be aware of what the matched label *AD2* represents.

To solve this issue, we can use our script *geturi.sh* to perform linking (aka entity disambiguation, entity mapping, normalization), i.e. find the classes in the disease ontology that may be represented by the recognized label. For example, to find what *AD2* represents, we can execute the following command:

```
$ echo "AD2" | ./geturi.sh doid.
 owl | ./getlabels.sh doid.
 owl
```

In this case, the result clearly shows that *AD2* represents the *Alzheimer disease*:

```
AD2
Alzheimer disease 2, late onset
Alzheimer disease associated
 with APOE4
Alzheimer disease-2
Alzheimer's disease 2
```

## Modified Labels

However, we may not be so lucky with the labels that were modified by our previous fixes in the lexicon. For example, we can test the case of *ATR*:

```
$ echo "ATR" | ./geturi.sh doid.
 owl
```

As expected, we received the warning that no URI was found:

```
XPath set is empty
```

An approach to address this issue may involve keeping a track of the original label in a lexicon using another file.

## Ambiguity

We may also have to deal with ambiguity problems where a label may represent multiple terms. For example, if we check how many classes the acronym *ATS* may represent:

```
$ echo "ATS" | ./geturi.sh doid.
 owl
```

We can see that it may represent two classes:

```
http://purl.obolibrary.org/obo/
 DOID_0050434
http://purl.obolibrary.org/obo/
 DOID_0110034
```

These two classes represent two distinct diseases, namely *Andersen-Tawil syndrome* and *X-linked Alport syndrome*, respectively.

We can also obtain their alternative labels by providing the two URI as standard input to the *getlabels.sh* script:

```
$ echo "http://purl.obolibrary.
 org/obo/DOID_0050434" | ./
 getlabels.sh doid.owl
$ echo "http://purl.obolibrary.
 org/obo/DOID_0110034" | ./
 getlabels.sh doid.owl
```

We will get the following two lists, both containing *ATS* as expected:

```
ANDERSEN CARDIODYSRHYTHMIC
 PERIODIC PARALYSIS
ATS
Andersen syndrome
LQT7
Long QT syndrome 7
Potassium-Sensitive
 Cardiodysrhythmic Type
Andersen-Tawil syndrome
ATS
nephropathy and deafness, X-
 linked
X-linked Alport syndrome
```

If we find a *ATS* mention in the text, the challenge is to identify which of the syndromes the mention refers to. For addressing this challenge, we may have to use advanced entity linking techniques that analyze the context of the text.

## Surrounding Entities

An intuitive solution is to select the class closer in terms of meaning to the others classes mentioned in the surrounding text. This assumes that entities present in a piece of text are somehow semantically related to each other, which is normally the case. At least the author assumed some type of relation between them, otherwise the entities would not be in the same sentence.

Let us consider the following sentence about genes and related syndromes from our text file *chebi\_27732\_sentences.txt* (on line 436):

```
... channel genes, KCNQ1 (LQT1),
 KCNH2 (LQT2), SCN5A (LQT3),
 KCNE1 (LQT5), and KCNE2 (LQT6
), along with KCNJ2 (Andersen
 -Tawil syndrome) and ...
```



Now assume that the label *Andersen-Tawil syndrome* been replaced by the acronym *ATS*:

```
... channel genes, KCNQ1 (LQT1),
 KCNH2 (LQT2), SCN5A (LQT3),
 KCNE1 (LQT5), and KCNE2 (LQT6
), along with KCNJ2 (ATS) and
```

Then, to identify the diseases in the previous sentence, we can execute the following command:

```
$ echo 'channel genes, KCNQ1 (
 LQT1), KCNH2 (LQT2), SCN5A
 (LQT3), KCNE1 (LQT5), and
 KCNE2 (LQT6), along with
 KCNJ2 (ATS) and' | grep -o
 -w -F -f diseases.txt
```

We have a list of labels that can help us decide which is the right class representing *ATS*:

```
LQT1
LQT2
LQT3
LQT5
LQT6
ATS
```

To find their URIs we can use the *geturi.sh* script:

```
$ echo 'channel genes, KCNQ1 (
 LQT1), KCNH2 (LQT2), SCN5A
 (LQT3), KCNE1 (LQT5), and
 KCNE2 (LQT6), along with
 KCNJ2 (ATS)
and' | grep -o -w -F -f
diseases.txt | ./geturi.sh
doid.owl
```

The only ambiguity is for *ATS* that returns two URIs, one representing the *Andersen-Tawil syndrome* (DOID:0050434) and the other representing the *X-linked Alport syndrome* (DOID:0110034):

```
http://purl.obolibrary.org/obo/
DOID_0110644
http://purl.obolibrary.org/obo/
DOID_0110645
http://purl.obolibrary.org/obo/
DOID_0110646
http://purl.obolibrary.org/obo/
DOID_0110647
http://purl.obolibrary.org/obo/
DOID_0110648
http://purl.obolibrary.org/obo/
DOID_0050434
http://purl.obolibrary.org/obo/
DOID_0110034
```

To decide which of the two URIs we should select, we can measure how close in meaning they are to the other diseases also found in the text.

## Semantic Similarity

Semantic similarity measures have been successfully applied to solve these ambiguity problems (Grego and Couto 2013). Semantic similarity quantifies how close two classes are in terms of semantics encoded in a given ontology (Couto and Lamurias 2019). Using the web tool Semantic Similarity Measures using Disjunctive Shared Information (DiShIn)<sup>13</sup>, we can calculate the semantic similarity between our recognized classes. For example, we can calculate the similarity between *LQTL* (DOID:0110644) and *Andersen-Tawil syndrome* (DOID:0050434) (see Fig. 5.6), and the similarity between *LQTL* and *X-linked Alport syndrome* (DOID:0110034) (see Fig. 5.7).

DiShIn

Not secure | labs.rd.ciencias.ulisboa.pt/dishin/

# DiShIn: Semantic Similarity Measures using Disjunctive Shared Information

**Ontology**

DO - Human Disease Ontology

**Entry 1**

DOID:0110644

Examples: CHEBI:31236, DOID:2841, GO:0000023 (or protein Q12345), HP:0000588, gold, RID16139, or ambulance-noun-1

**Entry 2**

DOID:0050434

Examples: CHEBI:3131, DOID:1324, GO:0000025 (or protein Q12346), HP:0001093, copper, RID16140, or motorcycle-noun-1

Submit

Measure	MICA/DiShIn	(Ex/In)trinsic	Similarity
Resnik	DiShIn	intrinsic	3.1715006566
Resnik	MICA	intrinsic	6.34300131319
Lin	DiShIn	intrinsic	0.376553538118
Lin	MICA	intrinsic	0.753107076235
JC	DiShIn	intrinsic	0.0952210062728
JC	MICA	intrinsic	0.240449173481

**Fig. 5.6** Semantic similarity between *LQTI* (DOID:0110644) and *Andersen-Tawil syndrome* (DOID:0050434) using the online tool DiShIn

The screenshot shows the DiShIn web application interface. The browser address bar indicates the URL is `labs.rd.ciencias.ulisboa.pt/dishin/`. The page title is "DiShIn: Semantic Similarity Measures using Disjunctive Shared Information".

**Ontology**  
 A dropdown menu is set to "DO - Human Disease Ontology".

**Entry 1**  
 Input field: `DOID:0110644`  
 Examples: CHEBI:31236, DOID:2841, GO:0000023 (or protein Q12345), HP:0000588, gold, RID16139, or ambulance-noun-1

**Entry 2**  
 Input field: `DOID:0110034`  
 Examples: CHEBI:3131, DOID:1324, GO:0000025 (or protein Q12346), HP:0001093, copper, RID16140, or motorcycle-noun-1

**Submit**

Measure	MICA/DiShIn	(Ex/In)trinsic	Similarity
Resnik	DiShIn	intrinsic	0.0
Resnik	MICA	intrinsic	0.0
Lin	DiShIn	intrinsic	0.0
Lin	MICA	intrinsic	-0.0
JC	DiShIn	intrinsic	0.0593651994576
JC	MICA	intrinsic	0.0593651994576

**Fig. 5.7** Semantic similarity between *LQTI* (DOID:0110644) and *X-linked Alport syndrome* (DOID:0110034) using the online tool DiShIn

## Measures

DiShIn provides the similarity values for three measures, namely Resnik, Lin and Jiang-Conrath (Resnik 1995; Lin et al. 1998; Jiang and Conrath 1997). The last two measures provide values between 0 and 1, and Jiang-Conrath is a

distance measure that is converted to similarity.

We can see that for all measures *LQT1* is much more similar to *Andersen-Tawil syndrome* than to *X-linked Alport syndrome*. Moreover, Jiang-Conrath's measure gives the only similarity value larger than zero for *X-linked Alport syndrome*, since it is a converted distance measure. We obtain similar results if we replace *LQT1* by *LQT2*, *LQT3*, *LQT5*, or *LQT6*. This means that by using semantic similarity we can identify *Andersen-Tawil syndrome* as the correct linked entity for the mention *ATS* in this text.

## DiShIn Installation

To automatize this process we can also execute DiShIn as a command line<sup>14</sup>, however we may need to install python (or python3) and SQLite<sup>15</sup>.

First, we need to install it locally using the `git` command line:

```
$ git clone git://github.com/
 lasigeBioTM/DiShIn.git
```

The `git` command automatically retrieves a tool from the GitHub<sup>16</sup> software repository.

If everything works fine, we should be able to see something like this in our display:

```
Cloning into 'DiShIn'...
...
Resolving deltas: 100% (255/255)
, done.
```

If the `git` command is not available, we can alternatively download the compressed file (zip), extract its contents and then move to the DiShIn folder:

```
$ curl -O -L https://github.com/
 lasigeBioTM/DiShIn/archive
 /master.zip
$ unzip master.zip
$ mv DiShIn-master DiShIn
```

The option `-L` enables the `curl` command to follow a URL redirection<sup>17</sup>. The equivalent long form to the `-L` option is `--location`.

We now have to copy the Human Disease Ontology in to the folder using the `cp` command, and then enter into the DiShIn folder:

```
$ cp doid.owl DiShIn/
$ cd DiShIn
```

## Database File

To execute DiShIn, we need first to convert the ontology file named *doid.owl* into a database (SQLite) file named *doid.db*:

```
$ python dishin.py doid.owl doid
.db http://purl.obolibrary
.org/obo/ http://www.w3.
org/2000/01/rdf-schema#
subClassOf ''
```

If the module rdflib is not installed, the following error will be displayed:  
ImportError: No module named

```
rdflib
```

We can try to install it<sup>18</sup>, but this will still take a few minutes to run.

Alternatively, we can download the latest database version:

```
$ curl -O http://labs.rd.
ciencias.ulisboa.pt/book/
doid.db
```

## DiShIn Execution

After being installed, we can execute DiShIn by providing the database and two classes identifiers:

```
$ python dishin.py doid.db
DOID_0110644 DOID_0050434
$ python dishin.py doid.db
DOID_0110644 DOID_0110034
```

The output of the first command will be the semantic similarity values between *LQTI* (DOID:0110644) and *Andersen-Tawil syndrome* (DOID:0050434):

```
Resnik DiShIn intrinsic
3.1715006566
Resnik MICA intrinsic
6.34300131319
Lin DiShIn intrinsic
0.376553538118
Lin MICA intrinsic
0.753107076235
JC DiShIn intrinsic
0.0952210062728
JC MICA intrinsic 0.240449173481
```

The output of the second command will be the semantic similarity values



between *LQTI* (DOID:0110644) and *X-linked Alport syndrome* (DOID:0110034):

```
Resnik DiShIn intrinsic 0.0
Resnik MICA intrinsic 0.0
Lin DiShIn intrinsic 0.0
Lin MICA intrinsic -0.0
JC DiShIn intrinsic
 0.0593651994576
JC MICA intrinsic
 0.0593651994576
```

In the end, we should not forget to return to our parent folder:

```
$ cd ..
```

Learning python<sup>19</sup> and SQL<sup>20</sup> is out of scope of this book, but if we do not intend to make any modifications the above steps should be quite simple to execute.

---

## Large Lexicons

The online tool MER is based on a shell script<sup>21</sup>, so it can be easily executed as a command line to efficiently recognize and link entities using large lexicons.

### MER Installation

First, we need to install it locally using the `git` command line:

```
$ git clone git://github.com/
 lasigeBioTM/MER.git
```

If everything works fine, we should be able to see something like this in our display:

```
Cloning into 'MER'...
...
Resolving deltas: 100%
(604/604), done.
```

If the `git` command is not available, we can alternatively download the compressed file (zip), and extract its contents:

```
$ curl -O -L https://github.com/
 lasigeBioTM/MER/archive/
 master.zip
$ unzip master.zip
$ mv MER-master MER
```

We now have to copy the Human Disease Ontology in to the data folder of MER, and then enter into the MER folder:

```
$ cp doid.owl MER/data/
$ cd MER
```

## Lexicon Files

To execute MER, we need first to create the lexicon files:

```
$ (cd data; ../
 produce_data_files.sh doid
 .owl)
```

This may take a few minutes to run. However, we only need to execute it once, each time we want to use a new version of the ontology. If we wait, the output will include the last patterns of each of the lexicon files.

Alternatively, we can download the lexicon files, and extract them into the data folder:

```
$ curl -O http://labs.rd.
 ciencias.ulisboa.pt/book/
 doid_lexicons.zip
$ unzip doid_lexicons.zip -d
 data/
```

We can check the contents of the created lexicons by using the `tail` command:

```
$ tail data/doid*
```

These patterns are created according to the number of words of each term. The output should be something like this:



```
==> data/doid_links.tsv <==
zika virus disease http://purl.
 obolibrary.org/obo/
 DOID_0060478
zikv congenital infection http
 ://purl.obolibrary.org/obo/
 DOID_0080180
zinacef allergy http://purl.
 obolibrary.org/obo/
 DOID_0040025
zinsser-cole-engman syndrome
 http://purl.obolibrary.org/
 obo/DOID_0070025
ziziphus mauritiana fruit
 allergy http://purl.
 obolibrary.org/obo/
 DOID_0060507
zlotogora-zilberman-tenenbaum
 syndrome http://purl.
 obolibrary.org/obo/
 DOID_0060773
zollinger-ellison syndrome http
 ://purl.obolibrary.org/obo/
 DOID_0050782
zoophilia http://purl.obolibrary
 .org/obo/DOID_9336
zoophobia http://purl.obolibrary
 .org/obo/DOID_600
zygomycosis http://purl.
 obolibrary.org/obo/DOID_8485
```

```
==> data/doid.txt <==
zika virus disease
zikv congenital infection
zinacef allergy
zinsser-cole-engman syndrome
ziziphus mauritiana fruit
 allergy
zlotogora-zilberman-tenenbaum
 syndrome
```

zollinger-ellison syndrome  
zoophilia  
zoophobia  
zygomycosis

==> data/doid\_word1.txt <==

xph  
xpid  
xpv  
xscid  
yaba  
yaws  
zaspopathy  
zoophilia  
zoophobia  
zygomycosis

==> data/doid\_word2.txt <==

yunis.varon syndrome  
zantac allergy  
zebrafish allergy  
zellweger syndrome  
zemuron allergy  
zika fever  
zinacef allergy  
zinsser.cole.engman syndrome  
zlotogora.zilberman.tenenbaum  
    syndrome  
zollinger.ellison syndrome

```

==> data/doid_words2.txt <==
yersinia infectious
yersinia pestis
yersinia pseudotuberculosis
y.linked monogenic
y.linked sertoli
y.linked spermatogenic
yolk sac
zika virus
zikv congenital
ziziphus mauritiana

==> data/doid_words.txt <==
y.linked spermatogenic failure 1
y.linked spermatogenic failure 2
yolk sac neoplasm
yolk sac tumor
yolk sac tumor of mediastinum
yolk sac tumor of the cns
zika virus congenital syndrome
zika virus disease
zikv congenital infection
ziziphus mauritiana fruit
allergy

```

## MER Execution

Now we are ready to execute MER, by providing each sentence from the file *chebi\_27732\_sen-ces.txt* as argument to its *get\_entities.sh* script.

```

$ cat ../chebi_27732_sentences.
 txt | tr -d "'" | xargs -I
 {} ./get_entities.sh '{}
 doid

```

We removed single quotes from the text, since they are special characters to the command line *xargs*. We should note that this is the *get\_entities.sh* script inside the MER folder, not the one we created before.

Now we will be able to obtain a large number of matches:

```

89 111 malignant hyperthermia
 http://purl.obolibrary.org/
 obo/DOID_8545
74 96 malignant hyperthermia
 http://purl.obolibrary.org/
 obo/DOID_8545
157 164 disease http://purl.
 obolibrary.org/obo/DOID_4
144 164 central core disease
 http://purl.obolibrary.org/
 obo/DOID_3529
13 20 disease http://purl.
 obolibrary.org/obo/DOID_4
47 55 myopathy http://purl.
 obolibrary.org/obo/DOID_423

```

...

The first two numbers represent the start and end position of the match in the sentence. They are followed by the name of the disease and its URI in the ontology.

We can also redirect the output to a TSV file named *diseases\_recognized.tsv*:

```

$ cat ../chebi_27732_sentences.
 txt | tr -d '"' | xargs -I
 {} ./get_entities.sh '{} '
 doid > ../
 diseases_recognized.tsv

```

We can now open the file in our spreadsheet application, such as LibreOffice Calc or Microsoft Excel (see Fig. 5.8).

	A	B	C	D
1	89	111	malignant hyperthermia	http://purl.obolibrary.org/obo/DOID_8545
2	74	96	malignant hyperthermia	http://purl.obolibrary.org/obo/DOID_8545
3	157	164	disease	http://purl.obolibrary.org/obo/DOID_4
4	144	164	central core disease	http://purl.obolibrary.org/obo/DOID_3529
5	13	20	disease	http://purl.obolibrary.org/obo/DOID_4
6	47	55	myopathy	http://purl.obolibrary.org/obo/DOID_423

**Fig. 5.8** The *diseases\_recognized.tsv* file opened in a spreadsheet application

Again, we should not forget to return to our parent folder in the end:

```

$ cd ..

```

---

## Further Reading

To know more about biomedical ontologies, the book entitled *Introduction to bio-ontologies* is an excellent option, covering most of the ontologies and computational techniques exploring them (Robinson and Bauer 2011).

Another approach is to read and watch the materials of the training course given by Barry Smith<sup>22</sup>.



**Open Access** This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.

---

## References

- Couto F, Lamurias A (2018) MER: a shell script and annotation server for minimal named entity recognition and linking. *J Cheminfo* 10(1):58  
[Crossref]
- Couto F, Lamurias A (2019) Semantic similarity definition. In: Ranganathan S, Nakai K, Schönbach C, Gribskov M (eds) *Encyclopedia of bioinformatics and computational biology*, vol 1. Oxford: Elsevier
- Couto FM, Campos LF, Lamurias A (2017) Mer: a minimal named-entity recognition tagger and annotation server. *Proc BioCreative* 5:130–7
- Grego T, Couto FM (2013) Enhancement of chemical entity identification in text using semantic similarity validation. *PloS one* 8(5):e62984  
[Crossref]
- Jiang JJ, Conrath DW (1997) Semantic similarity based on corpus statistics and lexical taxonomy. In: *Proceedings of the 10th research on computational linguistics international conference*, pp 19–33
- Lin D *et al* (1998) An information-theoretic definition of similarity. In: *Icml*, vol 98, pp 296–304. Citeseer
- Resnik P (1995) Using information content to evaluate semantic similarity in a taxonomy. In: *Proceedings of the 14th international joint conference on artificial intelligence*, vol 1, pp 448–453. Morgan Kaufmann Publishers Inc.
- Robinson PN, Bauer S (2011) *Introduction to bio-ontologies*. Chapman and Hall/CRC, Boca Raton  
[Crossref]

---

## Footnotes

1 <http://bioportal.bioontology.org/>

2 <http://www.obofoundry.org/>

3 <http://labs.rd.ciencias.ulisboa.pt/book/>

4 [http://purl.obolibrary.org/obo/DOID\\_66](http://purl.obolibrary.org/obo/DOID_66)

5 [http://purl.obolibrary.org/obo/CHEBI\\_26385](http://purl.obolibrary.org/obo/CHEBI_26385)

6 [http://purl.obolibrary.org/obo/CHEBI\\_27134](http://purl.obolibrary.org/obo/CHEBI_27134)

7 [http://purl.obolibrary.org/obo/CHEBI\\_25435](http://purl.obolibrary.org/obo/CHEBI_25435)

8 [http://purl.obolibrary.org/obo/RO\\_0000087](http://purl.obolibrary.org/obo/RO_0000087)

9 <https://en.wikipedia.org/wiki/Recursion>

10 <https://en.wikipedia.org/wiki/Stemming>

11 <https://en.wikipedia.org/wiki/UTF-8>

12 <https://en.wikipedia.org/wiki/ASCII>

13 <http://labs.rd.ciencias.ulisboa.pt/dishin/>

14 <https://github.com/lasigeBioTM/DiShIn>

15 `apt install python sqlite3` or `apt install python3 sqlite3`

16 <https://en.wikipedia.org/wiki/GitHub>

17 [https://en.wikipedia.org/wiki/URL\\_redirection](https://en.wikipedia.org/wiki/URL_redirection)

18 <https://github.com/RDFLib/rdfliib>

19 <https://www.w3schools.com/python/>

20 <https://www.w3schools.com/sql/>

21 <https://github.com/lasigeBioTM/MER>

22 [http://ontology.buffalo.edu/smith/IntroOntology\\_Course.html](http://ontology.buffalo.edu/smith/IntroOntology_Course.html)

# References

Allen G, Owens M (2011) The definitive guide to SQLite. Books for professionals by professionals. Apress, Berkeley

Angermueller C, Pärnamaa T, Parts L, Stegle O (2016) Deep learning for computational biology. *Mol Syst Biol* 12(7):878  
[\[PubMed\]](#)[\[PubMedCentral\]](#)

Aramaki E, Maskawa S, Morita M (2011) Twitter catches the flu: detecting influenza epidemics using twitter. In: Proceedings of the conference on empirical methods in natural language processing. Association for Computational Linguistics, pp 1568–1576

Aras H, Hackl-Sommer R, Schwantner M, Sofean M (2014) Applications and challenges of text mining with patents. In: *IPaMin@ KONVENS*

Ashburner M, Ball CA, Blake JA, Botstein D, Butler H, Cherry JM, Davis AP, Dolinski K, Dwight SS, Eppig JT *et al* (2000) Gene ontology: tool for the unification of biology. *Nat Genet* 25(1):25  
[\[PubMed\]](#)[\[PubMedCentral\]](#)

Baker J, Milligan I (2014) Counting and mining research data with unix. Technical report, The Editorial Board of the Programming Historian

Barros M, Couto FM (2016) Knowledge representation and management: a linked data perspective. *Yearb Med Inform* 25(1):178–183

Blumenthal D, Tavenner M (2010) The meaningful use regulation for electronic health records. *N Engl J Med* 363(6):501–504

Borst W, Borst W (1997) Construction of engineering ontologies for knowledge sharing and reuse. Ph.D. thesis, University of Twente

Campos L, Pedro V, Couto F (2017) Impact of translation on named-entity recognition in radiology texts. *Database* 2017:bax064

Canese K (2006) Pubmed celebrates its 10th anniversary. *NLM Tech Bull* 352:e5

Ching T, Himmelstein DS, Beaulieu-Jones BK, Kalinin AA, Do BT, Way GP, Ferrero E, Agapow P-M, Zietz M, Hoffman MM *et al* (2018) Opportunities and obstacles for deep learning in biology and medicine. *J R Soc Interface* 15(141):20170387  
[\[PubMed\]](#)[\[PubMedCentral\]](#)

Cock PJ, Antao T, Chang JT, Chapman BA, Cox CJ, Dalke A, Friedberg I, Hamelryck T, Kauff F, Wilczynski B *et al* (2009) Biopython: freely available python tools for computational molecular biology and bioinformatics. *Bioinformatics* 25(11):1422–1423  
[\[PubMed\]](#)[\[PubMedCentral\]](#)

Cook CE, Bergman MT, Cochrane G, Apweiler R, Birney E (2017) The european bioinformatics institute in 2017: data coordination and integration. *Nucleic Acids Res* 46(D1):D21–D29  
[\[PubMed\]](#)[\[PubMedCentral\]](#)

Coordinators NR (2018) Database resources of the national center for biotechnology information. *Nucleic Acids Res* 46(Database issue):D8



- Couto F, Lamurias A (2018) MER: a shell script and annotation server for minimal named entity recognition and linking. *J Cheminfo* 10(1):58
- Couto F, Lamurias A (2019) Semantic similarity definition. In: Ranganathan S, Nakai K, Schönbach C, Gribskov M (eds) *Encyclopedia of bioinformatics and computational biology*, vol 1. Oxford: Elsevier
- Couto FM, Campos LF, Lamurias A (2017) Mer: a minimal named-entity recognition tagger and annotation server. *Proc BioCreative* 5:130–7
- Couto FM, Silva MJ, Lee V, Dimmer E, Camon E, Apweiler R, Kirsch H, Rebholz-Schuhmann D (2006) GOAnnotator: linking protein go annotations to evidence text. *J Biomed Discov Collab* 1(1):19  
[\[PubMed\]](#)[\[PubMedCentral\]](#)
- Degtyarenko K, De Matos P, Ennis M, Hastings J, Zbinden M, McNaught A, Alcántara R, Darsow M, Guedj M, Ashburner M (2007) ChEBI: a database and ontology for chemical entities of biological interest. *Nucleic Acids Res* 36(suppl\_1):D344–D350  
[\[PubMed\]](#)[\[PubMedCentral\]](#)
- Doms A, Schroeder M (2005) GoPubMed: exploring pubmed with the gene ontology. *Nucleic Acids Res* 33(suppl\_2):W783–W786  
[\[PubMed\]](#)[\[PubMedCentral\]](#)
- Ferreira JD, Inácio B, Salek RM, Couto FM (2017) Assessing public metabolomics metadata, towards improving quality. *J Integr Bioinform* 14(4):67–72
- Forta B (2018) *Learning regular expressions*. Addison-Wesley Professional, Boston
- Gentleman RC, Carey VJ, Bates DM, Bolstad B, Dettling M, Dudoit S, Ellis B, Gautier L, Ge Y, Gentry J *et al* (2004) Bioconductor: open software development for computational biology and bioinformatics. *Genome Biol* 5(10):R80  
[\[PubMed\]](#)[\[PubMedCentral\]](#)
- Grego T, Couto FM (2013) Enhancement of chemical entity identification in text using semantic similarity validation. *PloS one* 8(5):e62984  
[\[PubMed\]](#)[\[PubMedCentral\]](#)
- Gruber TR (1993) A translation approach to portable ontology specifications. *Knowl Acquis* 5(2):199–220
- Haines N (2017) *Beginning Ubuntu for Windows and Mac users: start your journey into free and open source software*. Apress, Berkeley
- Hersh W (2008) *Information retrieval: a health and biomedical perspective*. Springer Science & Business Media, New York
- Hey T, Tansley S, Tolle KM *et al* (2009) *The fourth paradigm: data-intensive scientific discovery*, vol 1. Microsoft research Redmond, Redmond
- Holzinger A, Jurisica I (2014) Knowledge discovery and data mining in biomedical informatics: the future is in integrative, interactive machine learning solutions. In: *Interactive knowledge discovery and data mining in biomedical informatics*. Springer, Heidelberg, pp 1–18
- Holzinger A, Schantl J, Schroettner M, Seifert C, Verspoor K (2014) Biomedical text mining: state-of-the-art, open problems and future challenges. In: Holzinger A, Jurisica I (eds) *Interactive knowledge discovery*

and data mining in biomedical informatics. Springer, Heidelberg, pp 271–300

Hunter L, Cohen KB (2006) Biomedical language processing: what's beyond pubmed? *Mol Cell* 21(5):589–594

[\[PubMed\]](#)[\[PubMedCentral\]](#)

Jensen PB, Jensen LJ, Brunak S (2012) Mining electronic health records: towards better research applications and clinical care. *Nat Rev Genet* 13(6):395

Jiang JJ, Conrath DW (1997) Semantic similarity based on corpus statistics and lexical taxonomy. In: *Proceedings of the 10th research on computational linguistics international conference*, pp 19–33

Jurafsky D, Martin JH (2014) *Speech and language processing*, vol 3. Pearson, London

Kleene SC (1951) Representation of events in nerve nets and finite automata. Technical report, Rand Project Air Force, Santa Monica

Krallinger M, Rabal O, Lourenço A, Oyarzabal J, Valencia A (2017) Information retrieval and text mining technologies for chemistry. *Chem Rev* 117(12):7673–7761

Lamurias A, Couto F (2019) Text mining for bioinformatics using biomedical literature. In: Ranganathan S, Nakai K, Schönbach C, Gribskov M (eds) *Encyclopedia of bioinformatics and computational biology*, vol 1. Elsevier, Oxford

Lamurias A, Ferreira JD, Clarke LA, Couto FM (2017) Generating a tolerogenic cell therapy knowledge graph from literature. *Front Immunol* 8:1656

[\[PubMed\]](#)[\[PubMedCentral\]](#)

Leonelli S (2016) *Data-centric biology: a philosophical study*. University of Chicago Press, Chicago

Lesk A (2014) *Introduction to bioinformatics*. Oxford University Press, Oxford

Li W, Cowley A, Uludag M, Gur T, McWilliam H, Squizzato S, Park YM, Buso N, Lopez R (2015) The embl-ebi bioinformatics web and programmatic tools framework. *Nucleic Acids Res* 43(W1):W580–W584

[\[PubMed\]](#)[\[PubMedCentral\]](#)

Lin D *et al* (1998) An information-theoretic definition of similarity. In: *Icml*, vol 98, pp 296–304. Citeseer

Lu Z (2011) PubMed and beyond: a survey of web tools for searching biomedical literature. *Database* 2011:baq036

[\[PubMed\]](#)[\[PubMedCentral\]](#)

McGuinness DL, Van Harmelen F *et al* (2004) OWL web ontology language overview. *W3C Recommendation* 10(10):2004

Nosek BA, Alter G, Banks GC, Borsboom D, Bowman SD, Breckler SJ, Buck S, Chambers CD, Chin G, Christensen G *et al* (2015) Promoting an open research culture. *Science* 348(6242):1422–1425

[\[PubMed\]](#)[\[PubMedCentral\]](#)

Ong E, Xiang Z, Zhao B, Liu Y, Lin Y, Zheng J, Mungall C, Courtot M, Rutenberg A, He Y (2016) Ontobee: a linked ontology data server to support ontology term dereferencing, linkage, query and integration. *Nucleic Acids Res* 45(D1):D347–D352

[\[PubMed\]](#)[\[PubMedCentral\]](#)

- Rawat S, Meena S (2014) Publish or perish: where are we heading? J Res Med Sci 19(2):87  
[PubMed][PubMedCentral]
- Rebholz-Schuhmann D, Kirsch H, Couto F (2005) Facts from text—is text mining ready to deliver? PLoS Biol 3(2):e65  
[PubMed][PubMedCentral]
- Resnik P (1995) Using information content to evaluate semantic similarity in a taxonomy. In: Proceedings of the 14th international joint conference on artificial intelligence, vol 1, pp 448–453. Morgan Kaufmann Publishers Inc.
- Richardson L, Ruby S (2008) RESTful web services. O'Reilly Media, Inc., Sebastopol
- Ritchie DM (1971) Unix programmer's manual. Technical report, Technical report Bell
- Robinson PN, Bauer S (2011) Introduction to bio-ontologies. Chapman and Hall/CRC, Boca Raton
- Schriml LM, Mitra E, Munro J, Tauber B, Schor M, Nickle L, Felix V, Jeng L, Bearer C, Lichenstein R *et al* (2018) Human disease ontology 2018 update: classification, content and workflow expansion. Nucleic Acids Res 47:D955–D962  
[PubMed][PubMedCentral]
- Schuemle MJ, Weeber M, Schijvenaars BJ, van Mulligen EM, van der Eijk CC, Jelier R, Mons B, Kors JA (2004) Distribution of information in biomedical abstracts and full-text publications. Bioinformatics 20(16):2597–2604
- Shah, P. K., Perez-Iratxeta, C., Bork, P., & Andrade, M. A. (2003). Information extraction from full text scientific articles: where are the keywords? BMC Bioinformatics 4(1):20  
[PubMed][PubMedCentral]
- Shotts WE Jr (2012) The Linux command line: a complete introduction. No Starch Press, San Francisco
- Singhal A (2012) Introducing the knowledge graph: things, not strings. Official Google Blog 5. <https://googleblog.blogspot.com/2012/05/introducing-knowledge-graph-things-not.html>
- Smith B, Ashburner M, Rosse C, Bard J, Bug W, Ceusters W, Goldberg LJ, Eilbeck K, Ireland A, Mungall CJ *et al* (2007) The obo foundry: coordinated evolution of ontologies to support biomedical data integration. Nat Biotechnol 25(11):1251  
[PubMed][PubMedCentral]
- Spasic I, Ananiadou S, McNaught J, Kumar A (2005) Text mining and ontologies in biomedicine: making sense of raw text. Brief Bioinform 6(3):239–251  
[PubMed][PubMedCentral]
- Stajich JE, Block D, Boulez K, Brenner SE, Chervitz SA, Dagdigian C, Fuellen G, Gilbert JG, Korf I, Lapp H *et al* (2002) The bioperl toolkit: Perl modules for the life sciences. Genome Res 12(10):1611–1618  
[PubMed][PubMedCentral]
- Stephens ZD, Lee SY, Faghri F, Campbell RH, Zhai C, Efron MJ, Iyer R, Schatz MC, Sinha S, Robinson GE (2015) Big data: astronomical or genomics? PLoS Biol 13(7):e1002195  
[PubMed][PubMedCentral]
- Studer R, Benjamins VR, Fensel D *et al* (1998) Knowledge engineering: principles and methods. Data Knowl Eng 25(1):161–198

Styler IV WF, Bethard S, Finan S, Palmer M, Pradhan S, de Groen PC, Erickson B, Miller T, Lin C, Savova G *et al* (2014) Temporal annotation in the clinical domain. *Trans Assoc Comput Ling* 2:143

Tomczak A, Mortensen JM, Winnenburger R, Liu C, Alessi DT, Swamy V, Vallania F, Lofgren S, Haynes W, Shah NH *et al* (2018) Interpretation of biological experiments changes with evolution of the gene ontology and its annotations. *Sci Rep* 8(1):5115  
[\[PubMed\]](#)[\[PubMedCentral\]](#)

Wei C-H, Kao H-Y, Lu Z (2013) PubTator: a web-based text mining tool for assisting biocuration. *Nucleic Acids Res* 41(W1):W518–W522  
[\[PubMed\]](#)[\[PubMedCentral\]](#)

Wu D, Fung P (1994) Improving chinese tokenization with linguistic filters on statistical lexical acquisition. In: *Proceedings of the 4th conference on applied natural language processing*

Yeh A, Hirschman L, Morgan A (2003) Evaluation of text data mining for database curation: lessons learned from the KDD challenge cup. *Bioinformatics* 19(1):i331–i339

---

## Index

### A

Ancestors

### B

Bibliographic databases

Bioinformatics

Biomedical data repositories

Biomedical literature

### C

Chemical entities of biological interest (ChEBI)

Client uniform resource locator (cURL)

Command line tools

Comma-separated values (CSV)

Controlled vocabularies

### D

Data

    extraction

    filtering

    selection

Directed acyclic graphs (DAG)

Disease Ontology (DO)

## **E**

Entity

linking

European bioinformatics institute (EBI)

Evaluation metrics

Extensible markup language (XML)

## **L**

Lexicons

## **N**

Named-entity recognition (NER)

Natural language processing (NLP)

## **O**

Ontologies

Ontology elements

Open biomedical ontologies (OBO)

## **P**

Pattern matching

Pattern scanning

Programmatic access

## **R**

Recursion

Regular expressions

Relation extraction

## **S**

Semantics

Semantic similarity

Shell scripting

Spreadsheet applications

String matching

## **T**

Tab-separated values (TSV)

Terminal application

Text files

Text mining

Tokenization

## **U**

Uniform resource identifier (URI)

UniProt citations service

Unix shell

## **W**

Web ontology language (OWL)

Web retrieval

Word matching

## **X**

XML

*See* Extensible markup language (XML)

XML path language (XPath)